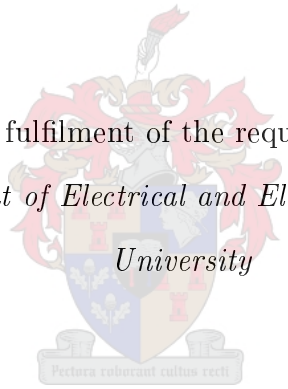
Parallelising Inference on Cluster Graphs

Author: Cornelis Arie VERBURGH



Report submitted in partial fulfilment of the requirements for the degree *Master of Engineering* in the *Department of Electrical and Electronic Engineering* at *Stellenbosch University*

Supervisors:

Prof. JA DU PREEZ,

Mr. A BARNARD

December 2020

DECLARATION

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

VERKLARING

Deur hierdie tesis elektronies in te lewer, verklaar ek dat die geheel van die werk hierin vervat, my eie, oorspronklike werk is, dat ek die alleenouteur daarvan is (behalwe in die mate uitdruklik anders aangedui), dat reproduksie en publikasie daarvan deur die Universiteit Stellenbosch nie derdepartyregte sal skend nie en dat ek dit nie vantevore, in die geheel of gedeeltelik, ter verkryging van enige kwalifikasie aangebied het nie.

Date: 04/09/2020

Abstract

Cluster graphs (CGs), a technique used in machine learning, are computationally very complex and often are slow to converge to an answer. There are also various approaches that lead to convergence. Some approaches converge faster than others. However, finding the optimal approach is non-trivial. We investigated the use of a new parallel inference algorithm, comparing it to the current state of the art inference algorithms, such as parallel splash belief propagation and residual belief update. These were tested on several CGs, ranging from a simple sudoku solver to a PGM that does satellite image denoising.

The results from these tests were as follows: for satellite image denoising a 5.3 times speed-up was achieved, plateauing at 10 threads; for the sudoku solver the speedup ranged from 2.0 times to 4.0 times speed-up, normally plateauing around six threads. From the results it is clear that the algorithms perform better the more clusters are present. It is also important to note that hyperthreading affects the speed-up, as is shown by the reduction in CPU instructions per cycle the more threads are being used.

Opsomming

"Cluster graphs" (CGs), 'n tegniek wat gebruik word in masjienleer, verg ingewikkelde berekeninge en is dikwels stadig om na 'n antwoord te konvergeer. Daar is ook verskillende benaderings wat lei tot konvergensie. Sommige benaderings konvergeer vinniger as ander. Om die optimale benadering te vind is egter nie triviaal nie. Ons ondersoek die gebruik van 'n nuwe algoritme vir parallelle inferensie, en vergelyk dit met die huidige moderne inferensie-algoritmes soos "parallel splash belief propagation" en "residual belief propagation". Dit word op verskeie CG's getoets, wat wissel van 'n eenvoudige sudoku-oplosser tot 'n CG wat satellietbeelde ontruis.

Die resultate van hierdie toetse was as volg: vir die satellietbeeld ontruising was dit 5.3 maal bespoedig en het by 10 "threads" 'n plato bereik; vir die sudoku-oplosser het dit bespoedig met tussen 2.0 en 4.0 maal en het normaalweg 'n plato bereik rondom ses "threads". Van die resultate is dit duidelik dat die algoritmes beter vaar hoe meer "clusters" daar is. Dit is ook belangrik om op te merk dat "hyperthreading" die hoeveelheid wat die toetse bespoedig beïnvloed, soos gewys deur die vermindering in CPU instruksies per siklus hoe meer "threads" gebruik word.

Acknowledgements

I would like to thank the following people for the support they offered me with regards to the project:

- My supervisors, for the weekly meetings that helped me stay on track.
- The CSIR, for the bursary support they have given me.
- My parents, for their love and faith in me.

Contents

List of Figures

List of Tables

List of Abbreviations

PGM	**P**robabilistic **G**raphical **M**odel
FG	**F**actor **G**raph
CG	**C**luster **G**raph
BP	**B**elief **P**ropagation
BU	**B**elief **U**pdate
MP	**M**essage **P**assing
MS	**M**essage **S**cheduling
RBP	**R**esidual **B**elief **P**ropagation
RBU	**R**esidual **B**elief **U**pdate
SPush-MS	**S**ingle **Push** **M**essage **S**cheduling
MPush-MS	**M**ultiple **Push** **M**essage **S**cheduling
Pull-MS	**Pull** **M**essage **S**cheduling
Smart-MS	**Smart** **M**essage **S**cheduling
Split-MS	**Split** **M**essage **S**cheduling
PSBP	**P**arallel **S**plash **B**elief **P**ropagation
OS	**O**perating **S**ystem
CPU	**C**entral **P**rocessing **U**nit

List of Symbols

S	sepset
s	sepset belief
m	message
Nb	Neighbour
ϕ	factor
ψ	cluster
β	belief

1 Introduction

This chapter explains the goal of the work and the relevance thereof and some background to understand the work. It also gives a brief overview of the work.

1.1 Some background on probabilistic graphical models and cluster graphs and parallelising inference thereon

This section discusses some underlying concepts necessary to understand probabilistic graphical models (PGMs) and cluster graphs (CGs) and parallel inference thereon. Understanding this section will clarify the objective of this work described in Section 1.2, as well as the contributions shown in Section 1.3.

1.1.1 Probabilistic graphical model

A probabilistic graphical model (PGM) is a way to represent very large joint distributions in a way that is tractable to calculate. It breaks up a single joint distribution into many smaller conditional and marginal distributions in a graph structure [1, p. 3].

For example,

$$P(X,Y) = P(X|Y)P(Y) \tag{1.1}$$

where X is dependent on Y. A graphical representation is shown in Figure 1.1.

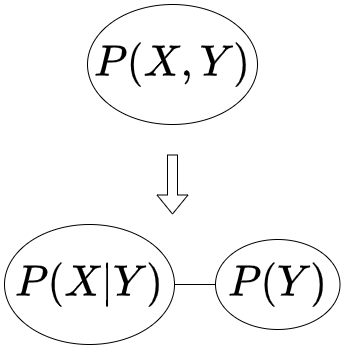


FIGURE 1.1: A graphical representation of a joint distribution, $P(X, Y)$, being split into a marginal distribution, $P(Y)$, and conditional distribution, $P(X|Y)$.

These graph structures allow for multiple calculations to be done simultaneously. We will elaborate on this in Section 1.1.6.

There are many uses for PGMs. They can be used to infer biological cellular networks [2], do layer-finding in radar echograms [3], do predictive vegetation mapping [4], detect maritime threats [5], solve sudoku puzzles and denoise satellite images. In this research, solving sudoku puzzles and denoising satellite images are used to measure the effectiveness of this work, as explained in Chapter 5.

There are many types of PGMs. Some representations include bayes nets (BN), markov random fields (MRFs) and factor graphs (FGs) [6]. There are also cluster graphs (CGs). In this work we will be focusing solely on CGs which are explained in Section 1.1.5. In Figure 1.2 a simple example of a BN, a MRF and a FG is shown.

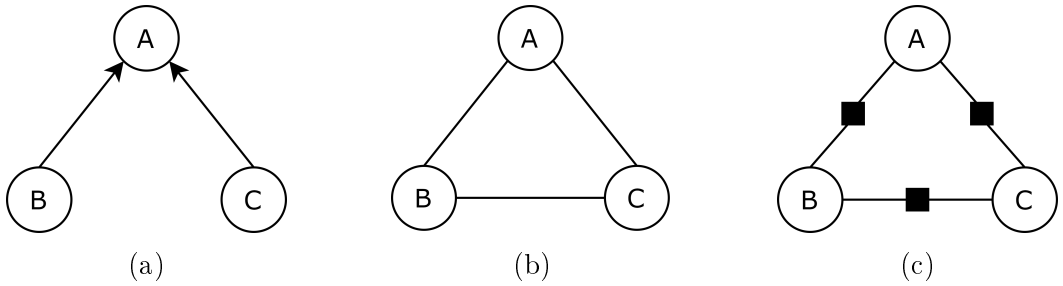


FIGURE 1.2: (a) A simple BN; (b) A simple MRF; (c) A simple FG.

1.1.2 Random variables

A random variable (RV) is a variable of which the value is uncertain. Their values can be discrete or continuous. An example of a discrete RV is called raining, with the value 0 representing that it is not raining and 1 representing that it is raining. Probabilities are linked to the two possible states of raining. This can also be extended to a combination of RVs [1, p. 20].

1.1.3 Factors

To understand a factor, we need to first explain what a potential is. A **potential** is a function of one or more RVs that must always be greater than zero, $\phi(x_1, ..., x_n) \geq 0$. A distribution is a special case of a potential, where the sum over all values for the RVs in the potential equals one, $\sum_{x_1,...,x_n} \phi(x_1, ..., x_n) = 1$ [6, p. 62]. Every factor represents a single potential (Figure 1.3 illustrates this). Note that we will only be looking at discrete factors.

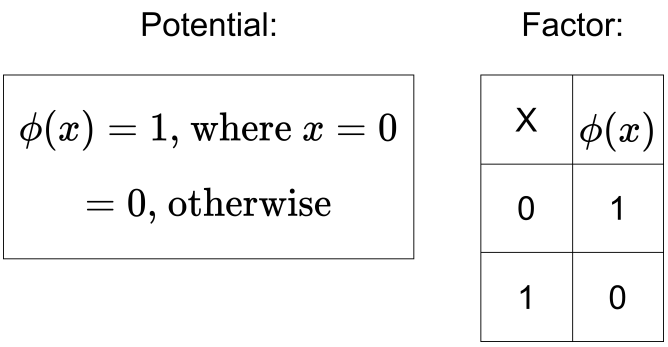


FIGURE 1.3: An illustration of a potential and a factor.

A **factor** describes the knowledge we have of a system. For instance, a factor can describe the relationship between the grass of a lawn being wet and whether it is raining (if the grass is wet, the likelihood that it is raining is higher). A factor can represent a joint distribution, or a conditional or marginal probability distribution over one or more RVs [1, p. 106].

1.1.4 Clusters: the nodes of a cluster graph

In the context of this work, a node refers to a cluster. A **cluster** is a collection of one or more factors multiplied together [1, p. 346]. Multiplication of factors are described in detail in Section 2.1. The specific factors that are multiplied together into clusters is a design choice. See Figure 1.4 for an illustration of this. This design affects the structure of the cluster graph (CG). CGs are explained below.

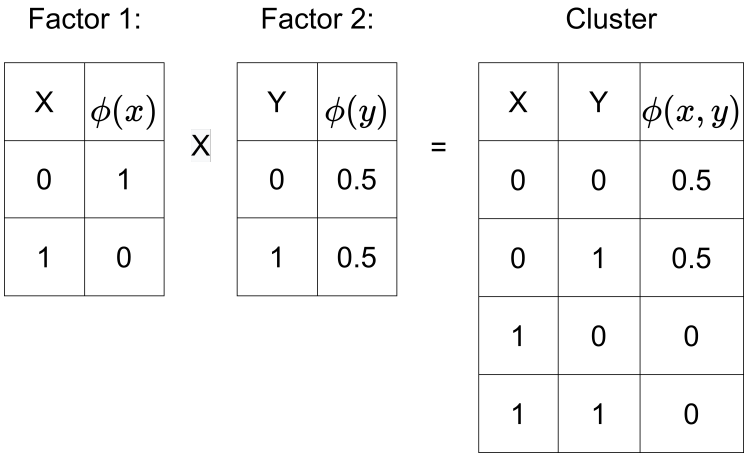


FIGURE 1.4: An illustration of how clusters are made up of one or more factors.

1.1.5 Linking clusters to form a cluster graph

An **edge** is a connection between two clusters in a CG, known as a **sepset**. This connection represents an overlap of one or more RVs between the two clusters it is connecting. If a cluster contains only a single RV, the edges connected to it will represent an overlap of only that RV. This overlap requires information to be sent between these clusters so that each cluster has more accurate probabilities for the RVs contained within it.

CGs will be explained in depth in Section 2.2.2.

1.1.6 Inference through message passing between the clusters of a cluster graph

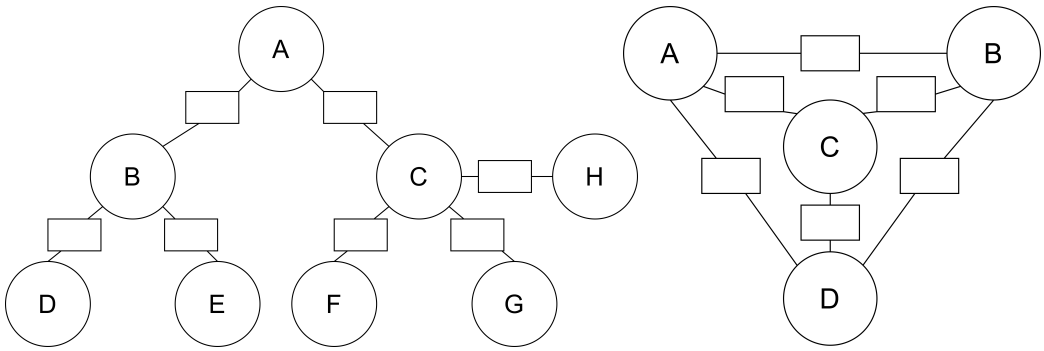
Message passing

A message between two clusters contains the probabilistic information of the cluster from which the message is being sent (the source cluster) about the RVs in the sepset between the source cluster and the cluster to which the message is being sent (the destination

cluster). This message is **absorbed** into the destination cluster to improve the accuracy of the information stored by the destination cluster. This is only a brief introduction to message passing. The details of how the messages are absorbed into the destination cluster will be discussed in more detail in Section 2.3.

Inference on tree-structured and loopy CGs

CGs come in many structures. In general, they can be in a tree structure or a loopy structure. The difference being that loopy CGs can contain multiple paths between clusters, also known as loops [1, p. 391], whereas tree structured CGs do not contain loops. These loops create dependency issues in the loopy CGs. Figure 1.5 shows an example of each structure.



(a) (b)

FIGURE 1.5: (a) A simple tree-structured CG; (b) A simple loopy CG.

To explain the dependency issues of a loopy CG consider the following example. When there are clusters arranged as in Figure 1.5(a) each cluster shares some RVs with the other clusters, none of them have all the information they need to pass exact messages to their neighbours. Each message is dependent on one or more other messages that are also incomplete. This prevents exact inference from taking place.

To obtain an approximately accurate message these messages have to be sent multiple times, each time containing more accurate information. Eventually the messages will converge to an approximate answer. The process of sending messages until convergence is called **inference**.

Message passing schedule

A message passing (MP) schedule is the order in which messages are sent in a CG. In loopy CGs there are multiple message orders that can be used to reach convergence [6, p. 84]. The order in which messages are sent determines the number of messages sent before convergence, as well as the accuracy of the convergence [1, p. 408].

Parallel message passing schedule

Inference can be done in parallel by sending multiple messages simultaneously. This work will focus on utilising parallel MP schedules to accelerate inference. The designed MP schedules are discussed in Chapter 4.

An important concept to note before using parallel MP schedules is that of **memory locks**. These memory locks are used to restrict access to specific memory locations to ensure that only one message is passed to each cluster at any given time. This causes some messages to wait for other messages to be passed, thereby reducing the number of messages being sent simultaneously. This is one of the reasons parallelising inference on CGs effectively is non-trivial. The use of these locks will be discussed further in Chapter 4.

To do multiple calculations simultaneously, **hardware threads** are used. A hardware thread (henceforth referred to as a thread) is a single logical processor, which may share some resources with other logical processors [7]. We elaborate on the architecture of the

threads used in Section 3.2.4 We also describe the system we used for this work in Chapter 5.

1.1.7 Using message queues to create message passing schedules

Message residuals

A **message residual** is the distance between a message and the old message along the same path (e.g. m_{ij} and m'_{ij}) [8]. This distance can be calculated using the Kullback-Leibler divergence [9]. The distance is an indication of the magnitude of change that has occurred between the new and old message. Larger changes, theoretically, have more important information to pass, as is confirmed in [8], where their residual belief propagation algorithm outperforms previous algorithms. Therefore, message residuals are used to prioritise which messages in the CG get passed first.

Message queues

A good way to schedule messages for rapid convergence in the fewest number of messages possible is to use a priority queue [8]. The priority queue dictates the order in which messages are sent. We will call this our **message queue**. These messages are prioritised according to their message residuals.

To send messages, the message at the top of the message queue (with the highest message residual) is removed (popped) from the queue. When a new message is inserted to the message queue, we first confirm whether that message is already in the queue. If the message is already in the message queue, the message residual is updated and the message is moved up in the queue according to the new priority. If the message is not in the message queue, it is simply added according to its message residual priority.

1.2 Objective

The goal of this thesis is to create a generic parallel message passing schedule for CGs, that speeds up inference while also maintaining the accuracy of the result.

1.3 Contributions

- We created a technique that reduces the waiting time caused by locks. This technique checks whether a cluster connected to the message at the top of the message queue is busy before popping the message from the message queue. If a cluster is busy, the message is skipped. This is explained in Section 4.3.
- Split message scheduling (Split-MS) was found to be our best performing MP schedule. Split-MS splits the messages into two parts (marginalising and absorbing), exposing extra parallelism within. The marginalising of the source cluster of a message and the absorbing of that message into the destination cluster are added to the message queue separately. Breaking the message into its two parts allows a source cluster to be marginalised, while the destination cluster is still busy absorbing other messages. This is explained in Section 4.4.
- The sequential version of Split-MS is an improvement over the standard residual belief update algorithm (explained in 3.1.2) in both speed-up and number of messages sent before convergence. This is because a cluster can absorb more than one message before sending new messages from that cluster, increasing the amount of information being passed by the new messages.
- By introducing manager threads Split-MS was further improved. These manager threads have all the control over the message queue. This allows the worker threads

to only do message calculations (marginalising source clusters and absorbing messages into destination clusters). The message queue is no longer shared between threads, removing the waiting time of threads accessing the message queue. This is explained in Section 4.4.3.

1.4 Motivation and topicality of this work

Inference on PGMs can be time-consuming. With the clock speed of CPUs having reached a peak, the next step to improve on algorithms is to parallelise them. Therefore, parallelising message passing to do inference on PGMs improves the speed at which PGM applications execute.

1.5 Scope of this thesis

Figure 1.6 shows the scope that was provided for this thesis. We used the EMDW library (explained below) and created parallel message passing schedules to improve the number of messages sent before convergence and the speed-up of inference on general CGs. The algorithms created can be applied to any representation of PGMs, but were specifically tailored to the PGM representations available in the EMDW library (CGs).

The **EMDW library** is built in C++ and is used to build and do inference on PGMs.

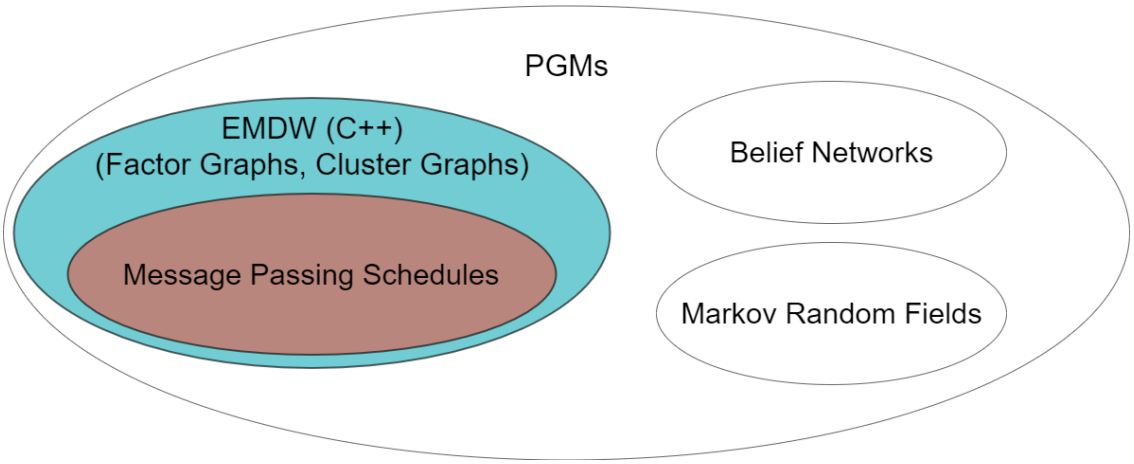


FIGURE 1.6: The scope provided for this work. The work is to be done using the EMDW library and C++ to create parallel message passing schedules.

1.6 Overview of this work

In this work, the inference of CGs in the EMDW library is parallelised with the aim to reduce the time it takes to reach convergence.

In Chapter 2, all the relevant information to understand the work that has not yet been discussed is explained. This includes a broader definition of CGs.

In Chapter 3, MP schedules that inspired the algorithms created for this work are discussed. We will compare this work to all these MP schedules. Parallelisation in C++ and some parallel programming concepts are also discussed.

There are 5 different parallel algorithms that are explained in Chapter 4. These are: single push message schedule (SPush-MS; explained in Section 4.1); multiple push message schedule (MPush-MS; explained in Section 4.1.2); pull message schedule (Pull-MS; explained in Section 4.2); smart message schedule (Smart-MS; explained in Section 4.3);

and split message schedule (Split-MS; explained in Section 4.4).

All the algorithms were tested on a sudoku solver and a satellite image denoiser. The results of measurements, the reasons for particular measurements and the methods used for measurements are explained in Chapter 5.

The results for comparing all the designed algorithms can be seen in Chapter 6. The results for each test showed that the speed-up of every algorithm plateaus at some point. It is important to remember that some threads share resources (as mentioned in Section 1.1.6), which affects the speed-up of every test. The best results were as follows:

- Satellite image denoising: 5.3 times speed-up; plateau at 10 threads.
- Sudoku solver with a cluster size of 7 RVs on puzzle 0: 3.3 times speed-up; plateau at 6 threads.
- Sudoku solver with a cluster size of 8 RVs on puzzle 0: 2.7 times speed-up; plateau at 2 threads.
- Sudoku solver with a cluster size of 7 RVs on puzzle 1: 4.0 times speed-up; plateau at 6 threads.
- Sudoku solver with a cluster size of 8 RVs on puzzle 1: 2.0 times speed-up; plateau at 5 threads.

The results are greatly affected by the number of clusters and edges present in the CGs that were used for testing. The higher the number of clusters and edges in the CG, the higher the speed-up. This is because the fewer clusters and edges a CG has, the more threads have to wait due to memory locks.

2 Probabilistic graphical models, cluster graphs and message passing

In this chapter all the relevant work on which this thesis has been built is explained. PGMs, cluster graphs (CGs) and message passing (MP) thereon are discussed.

2.1 Factors and operators that apply to them

Factors are described in Section 1.1.3. There are many operations that can be done on factors. These include normalising, marginalising, multiplication and division. These are explained here. All these operators extend to clusters as well, because clusters are products of one or more factors, as explained in Section 1.1.4.

Normalising In Figure 2.1, the normalisation of a small factor is shown. Normalising a factor means that the probabilities are scaled to sum to 1. This is done by dividing every probability by the sum of all the probabilities.

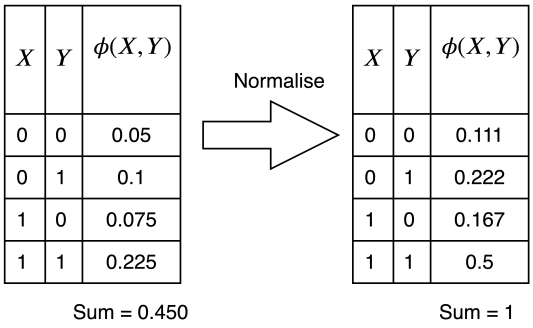


FIGURE 2.1: The factor, $\phi(X, Y)$, is normalised.

Marginalising In Figure 2.2, a simple marginalising example is shown with arbitrary values. The factor, $\phi(X, Y)$, is marginalised over the variable, Y, resulting in $\phi(X)$. The variable, Y, is removed from the factor by summing over all its values for each X.

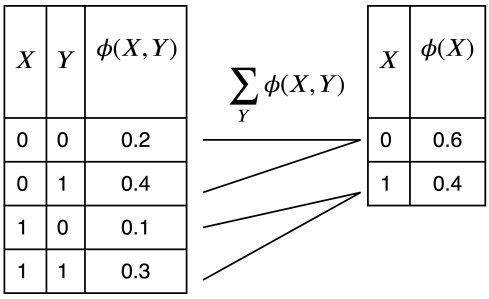


FIGURE 2.2: The factor, $\phi(X, Y)$, is marginalised over the variable, Y.

Multiplication Figure 2.3 illustrates a simple multiplication of two factors. Wherever the value of X coincides in either factor, the potentials get multiplied.

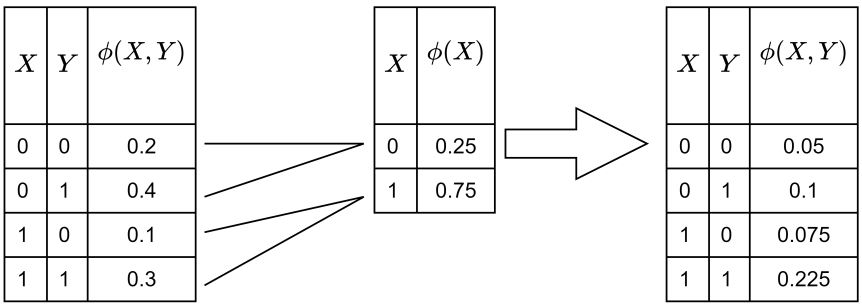


FIGURE 2.3: Two distributions are multiplied. Rows with matching values for the shared variable (X in this case) are combined by multiplying the corresponding potentials.

Division Figure 2.4 illustrates a simple division of two distributions. $\phi(X, Y)$ is divided by $\phi(X)$, except where $\phi(X)$ is equal to 0. Those values are made 0 instead, because they have been proven to be zero before.

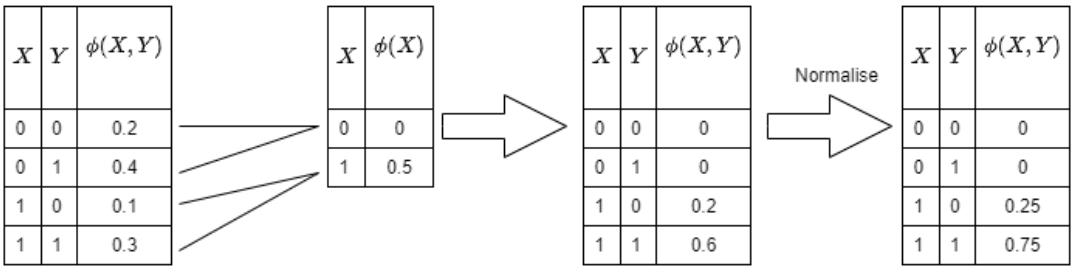


FIGURE 2.4: $\phi(X, Y)$ is divided by $\phi(X)$. In cases where division by zero seems called for, the result is set equal to zero instead.

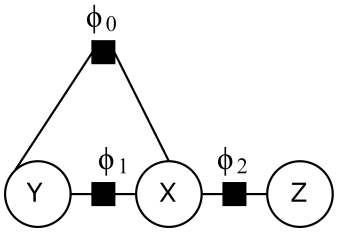
2.2 Different representations of PGMs

There are many ways to represent a PGM. The two representations discussed here are factor graphs (FGs) and cluster graphs (CGs).

2.2.1 Factor graphs

An FG is made up of nodes of RVs (circles) and factors (squares) and are connected by undirected edges. To illustrate how it works, an FG and its discrete factors are shown in

Figure 2.5 [6, p. 71].



(a) $\phi(X,Y,Z) = \phi_0(X,Y)\phi_1(X,Y)\phi_2(X,Z)$

X	Y	ϕ_0
0	0	0.2
0	1	0.3
1	0	0.4
1	1	0.1

(b) $\phi_0(X,Y)$

X	Y	ϕ_1
0	0	0.25
0	1	0.4
1	0	0.1
1	1	0.25

(c) $\phi_1(X,Y)$

X	Z	ϕ_2
0	0	0.05
0	1	0.5
1	0	0.4
1	1	0.05

(d) $\phi_2(X,Z)$

FIGURE 2.5: An FG and its factors. (a) The FG; (b) A factor containing the RVs X and Y; (c) A factor containing the RVs X and Y; Another factor containing the RVs X and Z

To bring the graphical representation of an FG and its equation together, we can write the RVs as uniform distributions, as this will not affect the joint distribution. For example,

$$P(X,Y) = P(X,Y)P(X)_{\text{uniform}}P(Y)_{\text{uniform}} \tag{2.1}$$

This allows us to more easily convert FGs into cluster graphs, which leads us to our next section.

2.2.2 Cluster graphs

A CG contains clusters with undirected edges between them. Each cluster contains one or more factors. The edge between two clusters is called a sepset (represented as a square). A sepset contains some RVs that are shared between the two clusters adjacent to the sepset [1, p. 346].

Note that not all RVs that are shared between two clusters are always present in the sepset. This is due to the **running intersection property** (RIP). RIP stipulates that there should be one unique path for every RV between the clusters it is present in. There can be no disconnects and there can be no cyclical dependencies for each RV. If there are disconnects, information about the relevant RV cannot be shared between all the clusters in which it is present. If there are cyclical dependencies, a piece of information might get passed along this cycle back to the original cluster that sent the information. This can cause the information to be biased [6, p. 108].

An illustration of a CG can be seen in Figure 2.6.

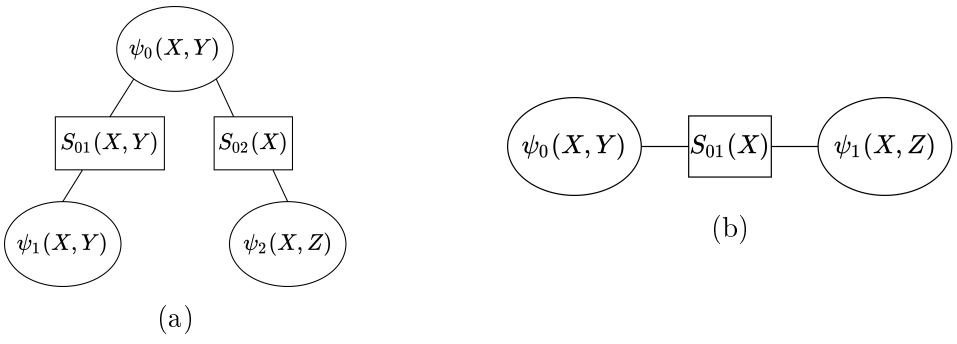


FIGURE 2.6: Two CGs that are both equivalent to Figure 2.5. All clusters share the RV X and clusters ψ_0 and ψ_1 share the RV Y. (a) Each cluster, ψ_i, coincides with each factor, ϕ_i, from 2.5. (b) ψ_0 encapsulates both factors ϕ_0 and ϕ_1, while ψ_1 only contains the factor ϕ_2.

Both the CGs in Figure 2.6 are equivalent to the FG in Figure 2.5. The difference between the two CGs is that Figure 2.6(a) has one cluster for each of the factors in Figure 2.5, while in Figure 2.6(b) ψ_0 contains both ϕ_0 and ϕ_1 from Figure 2.5.

We will be explaining the rest of this chapter in terms of CGs. However, the same concepts also apply to a factor graph.

2.3 Message calculation methods

In this section, we discuss the different techniques of doing belief propagation (BP). BP is how the messages in a CG are calculated and passed between clusters. First, we look at the original BP itself. Thereafter, we consider the belief update (direct approach; BU2) technique, as a slight improvement over BP, and finally the improved belief update (BU) technique, which we use in this work.

2.3.1 Belief propagation (sum-product)

BP is a way of passing information between clusters in a CG to do inference. It is a way for each cluster to obtain all the available information of the probabilities of its RVs. BP is also known as the Shafer-Shenoy algorithm [10].

BP works by sending messages between the clusters in a CG. In the context of a CG, a message between two clusters is calculated by multiplying (explained in Figure 2.3) all the incoming messages to the source cluster (except the one from the destination cluster) with the cluster distribution and summing (marginalising: this is explained in Figure 2.2)

over all the RVs not present in the sepset. It is also necessary to normalise (explained in Figure 2.1) the result, so as to prevent underflow or overflow [1, p. 352].

Initial messages The initial messages are calculated by only marginalising the source cluster for each message over all variables (except the variables present in the sepset along which the message is to be sent) and normalising the result.

The following equation explains how a message is calculated for BP:

$$m_{i \to j} = \sum_{\psi_i - S_{i,j}} \psi_i \cdot \prod_{k \in (\mathrm{Nb}_i - \{j\})} m_{k \to i} \tag{2.2}$$

A message sent from cluster i to cluster j is equal to cluster i multiplied by all the incoming messages to cluster i, except for the message from j to i, after which it is marginalised over all RVs in cluster i, except those in the sepset between i and j. The variables in ψ_i but not in the sepset, $S_{i,j}$, are denoted by $\psi_i - S_{i,j}$.

2.3.2 Belief update (direct approach)

The belief update (direct approach; let us call it BU2) algorithm is a variant of BP. BU2 is mathematically equivalent to BP [1, p. 364]. It does, however, require fewer calculations than BP, as is shown here.

BU2 makes use of division to replace the need for multiplying all the messages with the cluster distribution every time a message is calculated.

Beliefs To do BU2, the belief of a cluster is calculated once by multiplying all the incoming messages with the cluster distribution. To update the belief, it is divided by the old incoming message and multiplied by the new incoming message every time a message

is updated [1, p. 364].

The following is the equation for calculating the original belief of a cluster:

$$\beta_i = \psi_i \cdot \prod_{k \epsilon \mathrm{Nb}_i} m_{k \to i} \tag{2.3}$$

For cluster i, the original belief is calculated as the product of the cluster i distribution and neighbouring messages to i. The original beliefs use the original messages, which are calculated the same way as for BP (by marginalising only the source cluster of every message and normalising the result).

The following is the equation for calculating a new message for BU2:

$$m'_{i \to j} = \frac{\sum_{\psi_i - S_{i,j}} \beta_i}{m_{j \to i}} \tag{2.4}$$

The message from i to j is equal to the belief of cluster i, summed over all RVs present in cluster i, except for the RVs in the sepset of i and j, divided by the message from j to i.

The cluster beliefs are updated by absorbing the new message and dividing (explained in Figure 2.4) by the old message:

$$\beta'_i = \frac{\beta_i \cdot m'_{k \to i}}{m_{k \to i}} \tag{2.5}$$

where k is any adjacent cluster to i that has been updated.

2.3.3 Belief update

This algorithm further improves on the aforementioned BU2 algorithm, by exploiting the sepset beliefs. **Sepset beliefs** are the multiplication of the two messages adjacent to a sepset (one in each direction). Instead of dividing by the two messages separately, we instead divide by the sepset belief. BU is also known as the Lauritzen-Spiegelhalter algorithm [11].

Equation 2.6 shows the calculation of a sepset belief, s'_{ij}. The equation also shows how the sepset belief is equal to the marginal of the cluster belief, β_i, over all variables that are not in the sepset, S'_{ij}.

$$\begin{aligned} s'_{ij} &= m'_{i\to j} m_{j\to i} \\ &= \frac{\sum_{\psi_i - S_{i,j}} \beta_i}{m_{j\to i}} m_{j\to i} \\ &= \sum_{\psi_i - S_{i,j}} \beta_i \end{aligned} \tag{2.6}$$

Equation 2.7 shows the calculation of the new belief of cluster i. The new belief is equal to the old belief, β_i, multiplied by the new sepset belief, s'_{ij}, calculated in Equation 2.6, divided by the old sepset belief, s_{ij}.

$$\beta'_i = \frac{\beta_i \cdot s'_{ij}}{s_{ij}} \tag{2.7}$$

This version of BU is clearly the more efficient algorithm. Therefore, BU is the basis of all MP schedules shown in Chapter 4.

2.4 Push/ pull message passing

This section discusses two methods through which messages can be passed using a message queue (message queues are explained in Section 1.1.7).

Push message passing

In Figure 2.7, push message passing is illustrated.

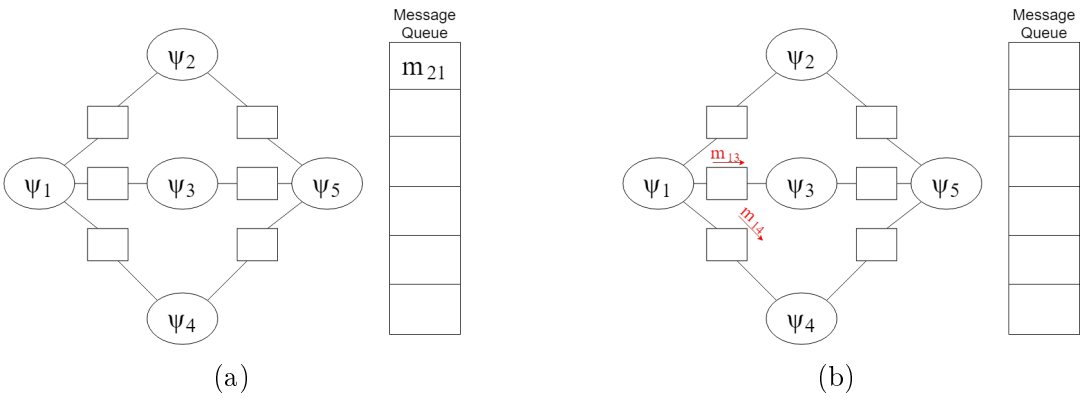


FIGURE 2.7: Here, push message passing is illustrated. The queue contains messages that were *previously* passed (i.e. at an earlier stage). When a message such as m_{21} is popped from the queue, the messages adjacent to it are then passed after which they get added to the queue.

The message m_{21} is at the top of the message queue. Therefore, it is popped first. m_{21} has already been passed using the BU method. When it is popped from the message queue, the messages adjacent to it (m_{13}; m_{14}) are passed [12].

Pull message passing

In Figure 2.8 pull message passing is illustrated.

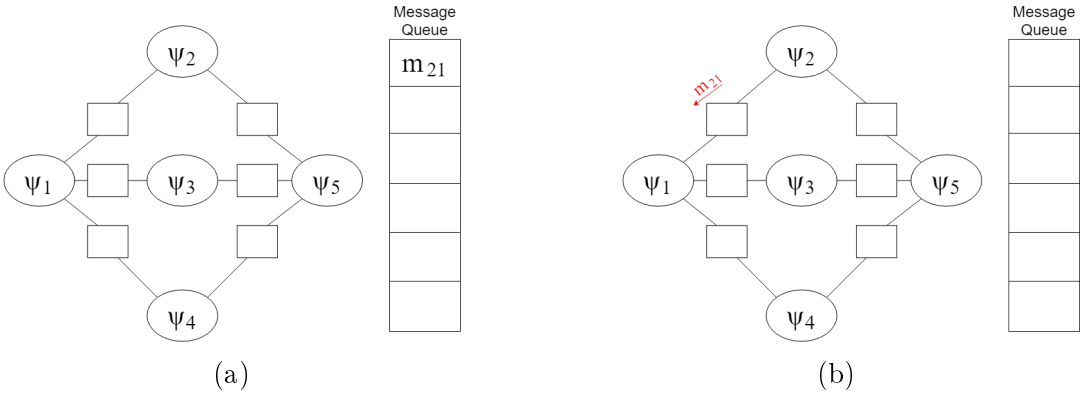


FIGURE 2.8: Here, pull message passing is illustrated. The messages in the queue are only scheduled to be passed (i.e. they are not passed yet). When a message such as m_{21} is popped from the queue, it is only *then* passed. The messages adjacent to it are then scheduled for later passing by adding them to the queue.

The message m_{21} is at the top of the message queue. It has not been passed at this point. After m_{21} is popped, it is passed [12].

Push/pull message passing compared

Pull message passing results in faster convergence with fewer messages being sent, compared to push message passing. This is due to the order in which messages are updated. Pull message passing always calculates the single most important message. Push message passing continues calculating all the messages adjacent to the message that was popped from the message queue. Any of these messages could add a more important message to the queue that should be dealt with first. However, push message passing finishes with the messages adjacent to the popped message before passing the potentially more important messages.

2.5 Summary

This chapter explained all the relevant work on which this thesis has been built. Cluster graphs (CGs), were discussed as the chosen representation for this thesis. Message passing (MP) on CGs was discussed. It was shown that the BU algorithm should be used to do message calculations, as it was the most efficient algorithm. Push and pull message passing were discussed. Both methods will be tested, as shown in Chapter 6 to compare them.

3 Literature study

This chapter discusses other message passing schedules that inspired the algorithms created in Chapter 4. It also discusses parallelising C++ code and some parallel programming concepts.

3.1 Message passing schedules

This section explains some of the fastest and most accurate message passing schedules to date. We show the two methods of residual belief propagation, called residual belief propagation and residual belief update. We also explain the parallel splash belief propagation algorithm.

3.1.1 Residual belief propagation

RBP uses BP (explained in Section 2.3.1) to pass messages in a cluster graph (CG). This can be applied to any representation of probabilistic graphical models, but the focus in this thesis is on CGs.

RBP uses message residuals (explained in Section 1.1.5) to add messages to a message queue (as explained in Section 1.1.7) in the order of highest priority to lowest priority. These messages can be passed using either push or pull message passing, explained in Section 2.4.

The order in which messages are sent using RBP allows for fewer messages to be sent before convergence, compared to older algorithms (such as round-robin).

3.1.2 Residual belief update

RBU is similar to RBP (explained in Section 3.1.1), but instead of using BP, it uses BU (explained in Section 2.3.2) to pass messages in a CG. These messages can be passed using either the push or pull message passing, explained in Section 2.4. The method we use is the pull method.

As explained in Section 2.3.2, BU requires fewer calculations than BP, while being mathematically equivalent. For this reason, RBU is faster than RBP.

3.1.3 Parallel splash belief propagation

PSBP is built on FGs (it can be adapted for CGs) and uses RBU to schedule which beliefs are chosen as the centre of a splash. From the centre of a splash, messages are added to the splash in breadth-first order, until the maximum size for the specific splash is reached. Afterwards, messages are sent from the leaf nodes of the splash to the centre and then back to the leaf nodes.

The size of a splash is determined by how large the belief residual at the centre node is. The maximum size of a splash (called its work) is normally chosen to be equal to n/p, where n is the number of vertices in the FG and p the number of processor threads used.

During a splash, every node that is updated is added to the priority queue with a priority equal to the belief residual (the distance between the old and new belief of the node).

3.1.4 Comparison between RBU and PSBP

We believe that, parallelised correctly, RBU can outperform PSBP. PSBP is compared to RBU in [13], although it is referred to as RBP. In all tests, RBU is outperformed by PSBP. This is due to the way [13] implements RBU.

The way RBU was coded in [13] was as a special case of PSBP, where the maximum work (explained in Section 3.1.3) allowed is set to 1. Therefore, all the overhead of PSBP slows this version of RBU down.

PSBP also schedules nodes to be updated instead of messages. Therefore, the RBU that [13] uses is not equivalent to the original RBU.

While belief residuals should result in faster convergence (as [13] theorises), clusters in a CG can be very large, causing it to become intractable to recalculate the belief residuals for every message passed. Therefore, using the PSBP algorithm on certain CGs should result in a slower speed-up compared to the original RBU.

Due to the reasons discussed above, we have decided to base the algorithms in this work on RBU, instead of PSBP.

We recognise that PSBP was published in 2009. However, all references to [13] only apply the PSBP algorithm. None of these papers attempt to improve on it. Therefore, [13] is the latest publication in this particular field of study and is relevant to this thesis.

3.2 Parallelising C++

In parallel programming, there are three contexts in which to parallelise code: shared memory; distributed memory; and graphical processing unit parallelisation. These are discussed here. We also explained why shared memory is the chosen method of parallelisation.

3.2.1 Distributed memory

Distributed memory means that every thread of the processor has its own memory space. The threads do not share memory. Distributed memory parallelisation allows threads to access memory without having to protect it using memory locks, because the threads never write to the same memory location [14, p. 7].

However, when threads communicate with each other in a distributed environment, the latency of communication is very high compared to accessing their local memory. This is due to the relevant memory being copied from one thread's memory to another [15, p. 8].

3.2.2 Shared memory

Shared memory means that all the threads of the processor have access to the same memory space [15, p. 8]. Therefore, shared memory parallelisation does not have the communication latency problem that distributed memory has.

However, the shared access to the memory space causes other problems. When more than one thread attempts to write to the same memory location, undefined behaviour may occur. This is called a **race condition** [16]. Therefore, locks are implemented to protect against these race conditions. These locks allow only a single thread to access a space

in memory at a time causing parts of the code to become sequential and reducing the possible parallelisation.

Another disadvantage of shared memory parallelism is the limiting of the memory modules. In shared memory parallelism, these memory modules cannot grow with the size of some problems as there is a limited number of hardware memory modules that can be added, while distributed memory can expand the number of hardware memory modules as needed [14, p. 7].

We have chosen to use shared memory parallelisation, because we believe much of the waiting time at shared memory locations can be prevented in the context of CGs. This can be done by adapting the schedule of the messages to prevent memory clashes.

3.2.3 Graphical processing unit parallelisation

We considered using graphical processing unit (GPU) off-loading for parallelisation. However, this presents problems of compatibility with the current state of the EMDW library (explained in Section 1.5), on which this work is built. The EMDW library uses C++ std::maps (as described in [17, p. 333]) to store most of the information of the CGs. Off-loading work to the GPU requires pointers to blocks of data (arrays), while C++ std::maps are stored randomly across memory [18, p. 26]. To convert these std::maps to arrays and off-load them to the GPU would be computationally very expensive. This could potentially outweigh the speed-up gained from the off-load.

To parallelise the operations on the factors themselves is not a general solution, because there are different types of factors (Discrete tables; Gaussian distributions; Dirichlet distributions) that require different operations. Therefore, this method of parallelisation is

not used [18, p. 42].

3.2.4 Understanding parallelisation

Amdahl's law

An important law to remember when parallelising is Amdahl's law. Amdahl's law states that there is a limit to the speed-up that can be gained by parallelising. This is because some parts of the process access shared memory, which causes those parts to be sequential. Equation 3.1 illustrates the law.

$$\text{Speedup}(f, n) = \frac{1}{(1-f) + \frac{f}{n}} \tag{3.1}$$

where f is the fraction of the process that can be parallelised and n is the number of threads used to parallelise [19]. This implies that, when n tends to infinity, the speed-up of the process is bound by Equation 3.2 [20].

$$\text{Speedup}(f, n) = \frac{1}{1-f} \tag{3.2}$$

Note that these equations do not account for the overhead caused by parallelising a process.

Hyper-threading

Hyper-threading is used in certain processor architectures to increase the number of threads available to processes. Processors with hyper-threading have two copies of the architectural state (all registers) on every core, but only one set of physical execution resources [21]. These execution resources can never be fully utilised by one thread. Allowing two threads to access the same execution resources increases the utilisation of these

resources. However, these two threads are now competing for the same resources. Therefore, using these two threads on one core does not necessarily increase the speed-up of a perfectly parallelisable program by a factor of 2 times. The utilisation of these resources can be quantified by measuring the instructions per core cycle [22].

Hyper-threading can lead to a bottleneck caused by the memory bandwidth of a system. Also, the increased communication demand, due to the higher number of threads, causes competition at resources such as the host-channel adapter (HCA) chips and Infiniband (IB) switches [22].

The two threads per core also share a cache line. The competition on the cache line can lead to more cache misses, causing threads to be less efficient [23].

3.2.5 OpenMP

OpenMP is a shared memory parallelisation interface, that handles the scheduling of parallel threads. To understand its usefulness, a brief explanation of the ways OpenMP can lock memory is explained.

To protect against race conditions that can show up in the shared memory context, the following OpenMP constructs are used: critical, atomic and locks.

Critical A critical region is used to protect a section of code in which shared memory is accessed. Only one thread can enter a critical region at a time. If this method is used incorrectly, it can cause large sequential regions in the program [24, p. 87].

Atomic The atomic construct is similar to the critical region. However, an atomic region can only protect a single atomic operation, such as incrementing or decrementing an integer value. It has less overhead than an equivalent critical region [24, p. 90].

Locks A lock is more flexible than a critical or atomic region. It can lock any block of memory, whether it is a single element in an array or multiple elements. This allows threads to access the same part of code in parallel for different parts in memory. However, a lock has additional overhead to initialise and destroy it. Therefore, a lock must be used multiple times to reduce the effect of the overhead [24, p. 93]. Using a lock multiple times does not imply that threads will wait at the lock every time they reach it. Preferably there should never be threads waiting at locks. The program should be written in a way to reduce the number of clashes at a lock.

These are by no means an exhaustive list of the possibilities of OpenMP. These are just the most important constructs used.

To parallelise C++, we decided to use OpenMP. It has most of the versatility of POSIX threads (PThreads), while having a simpler API. The code written using OpenMP is much simpler compared to the equivalent PThreads code. PThreads is also more prone to silent data corruption than OpenMP [25]. Therefore, OpenMP is the parallel programming interface used to design all the algorithms shown in Chapter 4.

4 Design of parallel message passing system

This chapter explains the different parallel message passing (MP) schedules created. All the algorithms explained in this chapter build on the sequential residual belief update (RBU) schedule and are built for cluster graphs (CGs). These new schedules take advantage of different parallelising techniques to gain the best possible speed-up.

4.1 Push parallelisation

Push message passing, as explained in Section 2.4, is when messages adjacent to a *previously* passed message are passed. There are two ways to parallelise push message passing. The messages inside a single push can be passed in parallel, or multiple pushes can happen in parallel. These two methods and their advantages and disadvantages are explained in this section.

4.1.1 Single push parallelisation

This method, called the single push message schedule (SPush-MS), parallelises a single push (described in Section 2.4). The scheduling of the messages is based on residual belief update (RBU), explained in Section 3.1.2. As with RBU, the message on the top of the queue (which has the highest priority) is popped. The messages that are adjacent to the popped message's destination cluster are then passed in parallel with one another. The

messages that were passed are added to the queue with a priority equal to the message residual.

Figure 4.1 illustrates this schedule. m_{21} is popped from the queue by the manager thread. The messages adjacent to the destination cluster of m_{21} (which are m_{13} and m_{14}) are then passed by two parallel threads. Subsequently, m_{13} and m_{14} are added to the queue by the two threads.

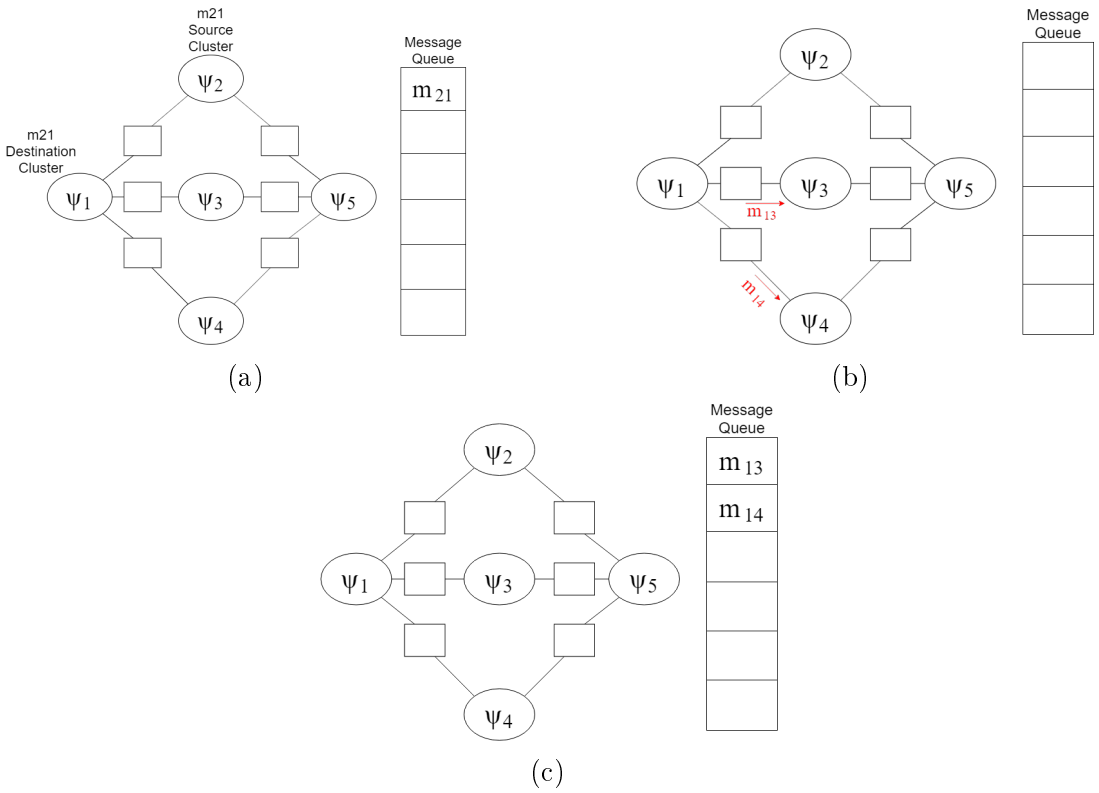


FIGURE 4.1: Two threads (thread 0 and 1) are used to do inference on this CG. (a) m_{21} is at the top of the message queue and is popped from the queue by thread 0. (b) The messages adjacent to m_{21} (m_{13} and m_{14}) are calculated in parallel by threads 0 and 1. (c) After m_{13} and m_{14} have been passed, they are added to the queue. The priority with which each message is added to the message queue is equal to the message residual.

Note that the two threads cannot access the message queue simultaneously. Therefore,

the first thread locks the message queue before inserting m_{13}, forcing the other thread to wait. After the first thread has inserted m_{13} into the queue it releases the lock and the second thread can lock the queue and insert m_{14}.

Advantages

Using this method prevents any memory clashes to occur at clusters. Threads always read from the same cluster, but never write to the same cluster. Thus, no locks on the clusters are required.

Disadvantages

Every time a message is popped from the message queue, the threads that pass the adjacent messages in parallel have to be opened. It also closes after the messages have been added to the message queue. This adds some overhead time, causing a slower speed-up.

SPush-MS performs poorly on CGs with low-degree clusters. This is due to the limited parallelisation it would allow. Considering Figure 4.1 again, if the inference on this CG is done using three parallel threads, only one or two of those threads work at any given time. Thus the speed-up of this technique is greatly influenced by the average degree of clusters in a given CG.

There is another factor affecting the speed-up negatively. Threads that receive smaller messages to pass have to idle while the other threads with larger messages finish their calculations.

4.1.2 Multiple push parallelisation

This method, called the multiple push message schedule (MPush-MS), calculates multiple pushes in parallel, using push message passing (described in Section 2.4). The scheduling of the messages is based on RBU, explained in Section 3.1.2. As with RBU, the message on the top of the queue (which has the highest priority) is popped first. However, multiple messages are popped. Each available thread is given one of these popped messages. Each thread loops through the messages adjacent to their popped message, passing these messages one at a time. To illustrate this concept, see Figure 4.2.

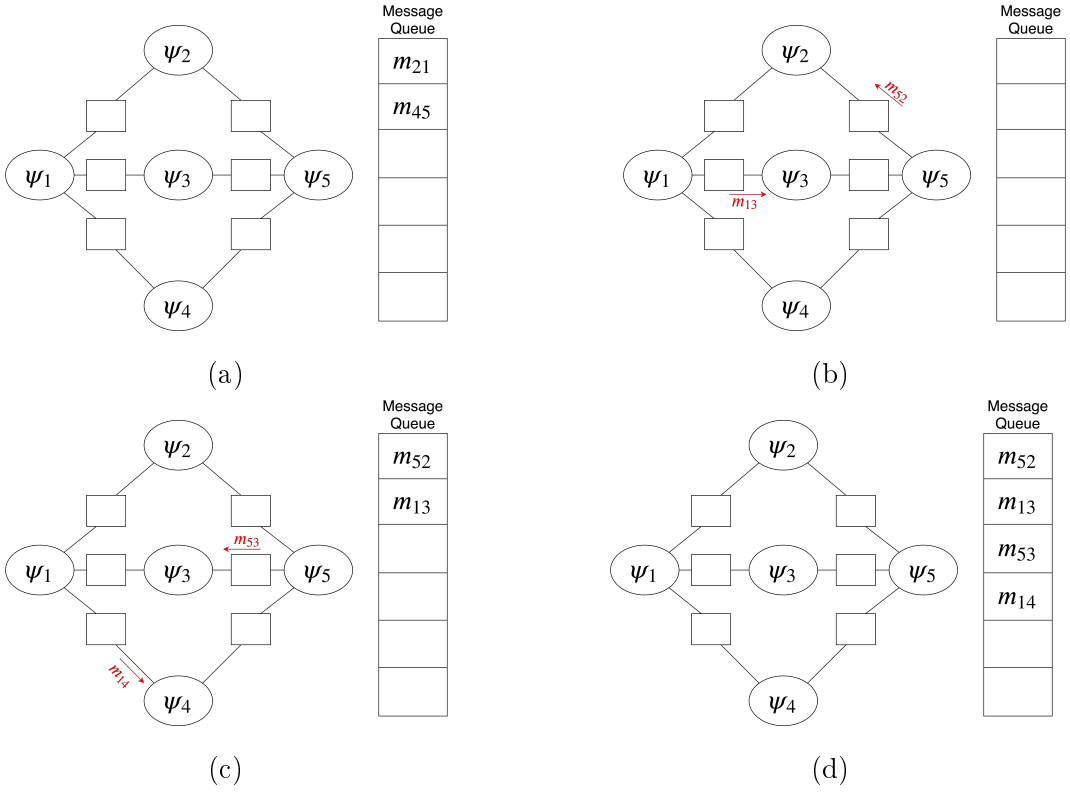


FIGURE 4.2: Two threads (thread 0 and 1) are used to do inference on this CG. (a) m_{21} and m_{45} are on the message queue and are popped from the queue by the master thread (thread 0). (b) and (c) The messages adjacent to m_{21} and m_{45} (m_{13} and m_{14}, and m_{52} and m_{53} respectively) are calculated in parallel by threads 0 and 1. Thread 0 first calculates m_{13} and then m_{14}, while thread 1 first calculates m_{52} and then m_{53}. (c) After each message has been passed, it is added to the queue. The priority with which each message is added to the message queue is equal to the message residual.

Advantages

Using MPush-MS means that threads do not wait for other threads to finish passing their messages before popping their next message.

Disadvantages

Memory clashes at clusters are possible (e.g. cluster 3 in Figure 4.2(b)). This is due to the fact that threads are not reading from the same cluster. This implies that it is

possible for them to try and write to the same cluster. Locks are added at each cluster (both at the source and destination clusters) to prevent the threads from writing to the same memory location. This causes some threads to wait for others when they want to write or read from the same cluster. It is important to lock the source clusters as well, because another thread might attempt to write to that cluster while it is being read from.

4.2 Pull parallelisation

This method, called the pull message schedule (Pull-MS), uses the pull message passing (described in Section 2.4). Instead of calculating the messages adjacent to the queued message's destination cluster, the queued message is the one that is calculated.

Figure 4.3 demonstrates the pull parallelisation on a CG. Thread 0 locks the message queue, pops m_{21} and unlocks the message queue. Thereafter, it can pass m_{21}. As soon as thread 0 releases the lock on the message queue, thread 1 can access the message queue. Thread 1 locks the message queue, pops m_{45} and unlocks the message queue. Thread 1 can now pass m_{45}.

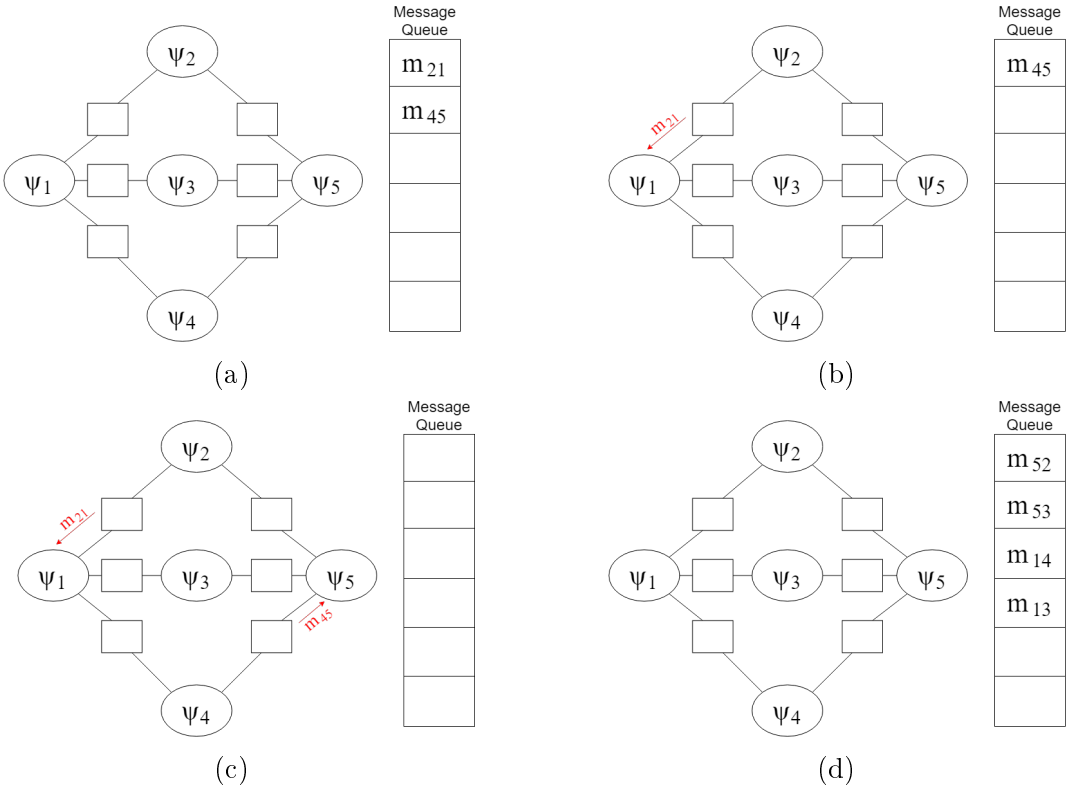


FIGURE 4.3: Two threads (threads 0 and 1) are doing inference on the displayed CG. (a) Thread 0 locks the message queue, pops m_{21} and unlocks the message queue. (b) While thread 0 passes m_{21}, thread 1 locks the message queue, pops m_{45} and unlocks the message queue. (c) Thread 0 locks the message queue to insert the messages adjacent to m_{21} and unlocks it afterwards. Thread 1 does the same with m_{45}. (d) The new messages have been added to the message queue.

This algorithm is the natural parallelisation of the RBU algorithm (explained in Section 3.1.2). PSBP (explained in Section 3.1.3) is compared to the natural parallelisation of RBU in [13]. Therefore, we can use Pull-MS to indirectly compare our algorithms to that of the PSBP algorithm.

4.2.1 Advantages

As explained in Section 2.4, pull message passing should result in faster convergence, compared to push message passing. This is because pull message passing always calculates

the single most important message, while push message passing does not. This implies that parallelising pull MP should also be faster than parallelising push MP.

4.2.2 Disadvantages

Thread clashes at clusters

This method is not reliant on the degree of connections between clusters in the same way as SPush-MS (explained in Section 4.1). It does not require high degree connections to work effectively. However, this method allows for thread clashes to occur.

A thread clash is defined as a point of contention between two hardware threads for shared memory. If the threads are allowed to access this memory location at the same time, undefined behaviour follows. To prevent this, OpenMP locks are put on each cluster. For a thread to access a cluster, the lock for that cluster is required. A thread cannot access a memory location if the lock thereof is already in use by another thread.

The higher the degree of connections of a cluster is, the more thread clashes occur. These thread clashes cause some threads to wait for other threads to finish their work. This wastes valuable time, reducing the overall speed-up gained.

Too many locks

Another drawback of this technique is the many locks required to prevent thread clashes. Having a lock on every cluster becomes a problem with large CGs. OpenMP locks require a lot of memory. Initialising and destroying these locks also adds some overhead.

4.3 Smart parallel message schedule

4.3.1 Single set locking

To reduce the memory usage that is required to create a lock on each cluster we replaced the locks with a set that contains all the clusters that are in use by a thread. When a thread pops a message from the queue it adds the two clusters that are adjacent to the message (the source and destination clusters) onto the busy set. If a different thread attempts to pop a message adjacent to either of these clusters the message is skipped. The next most important message on the queue is considered instead. This process continues until a message is found of which neither adjacent clusters are already in use. Remember that the clusters being read from also need to be locked, so that another thread cannot write to the cluster at the same time it is being read from.

In Figure 4.4, two threads are in use. Thread 0 has locked ψ_1 and ψ_2 and is passing m_{21}, while thread 1 is attempting to pop a message from the queue. Thread 1 cannot pop m_{13} from the queue, because it needs to read from ψ_1. Therefore, thread 1 steps over m_{13} and attempts to pop m_{45}. Seeing as m_{45} does not require access to either ψ_1 and ψ_2, it is popped. Subsequently, ψ_4 and ψ_5 are added to the busy set.

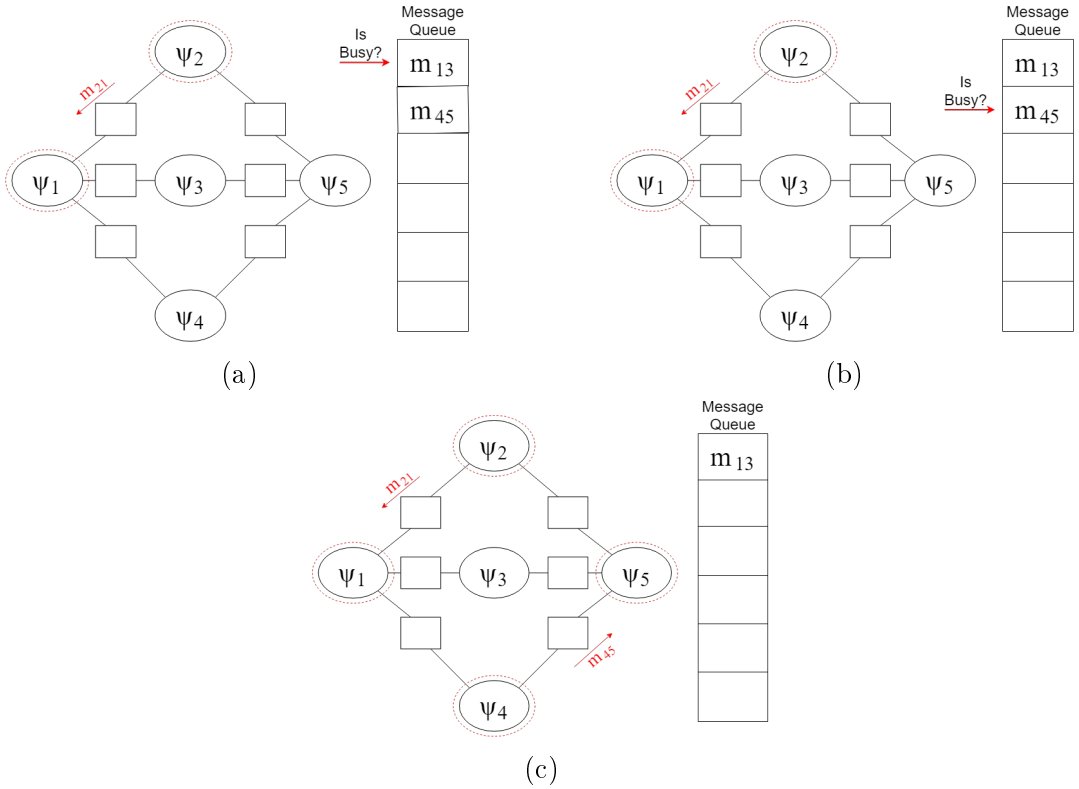


FIGURE 4.4: (a) Thread 0 passes m_{21}, while thread 1 attempts to pop m_{13}, however, ψ_1 is already busy. Therefore, m_{13} is stepped over. (b) Thread 1 attempts to pop m_{45} from the queue and succeeds. (c) Thread 1 adds ψ_4 and ψ_5 to the busy set and passes m_{45}.

4.3.2 Double set locking

Expanding on the single set locking idea, we introduced the use of two sets. One set contains the indices of all clusters that are busy being written to (busy write set) and another set is filled with all clusters that are busy being read from (busy read set). This allows us to refine which messages get popped from the queue.

To understand why this separation is important, remember the two main parts of how a message gets passed: marginalising the source cluster; and absorbing the message into the destination cluster (explained in Section 2.3.3). The destination cluster is being written

to. Therefore, only one thread should access this cluster at a time. However, the source cluster is being read from. Multiple threads can read from the same cluster at the same time as long as none of them are writing to this cluster. This allows multiple messages with the same source cluster to be calculated at the same time.

Without the busy read set, a thread would be able start writing to a cluster that is already being read from. This would cause undefined behaviour. A busy read set is therefore necessary.

4.3.3 Advantages

To illustrate the reduction of thread clashes due to the double set locking, see Figure 4.5. In this example, three threads are reading from the same cluster at the same time. This would not be possible using only the single set locking technique.

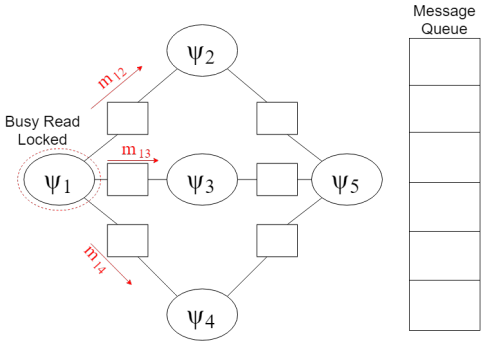


FIGURE 4.5: Three threads are reading from ψ_1 at the same time to pass their respective messages.

4.3.4 Disadvantages

While double set locking reduces the thread clashes, it does not eliminate it. This remains a problem near the end of convergence, when there are only a few messages left on the

queue. All the remaining messages near the end of convergence tend to share clusters. Therefore, only a few of the messages can be calculated at a time. This causes some threads to keep searching through the message queue for a message to pop until another thread finishes calculating its message and releasing the adjacent clusters.

Another bottleneck that this technique does not overcome occurs when a CG has a high degree of connection. For example, let us consider Figure 4.6 below. It is possible that three messages share the same source or destination cluster. Therefore, only one of those messages can be calculated at a time, wasting valuable processing time.

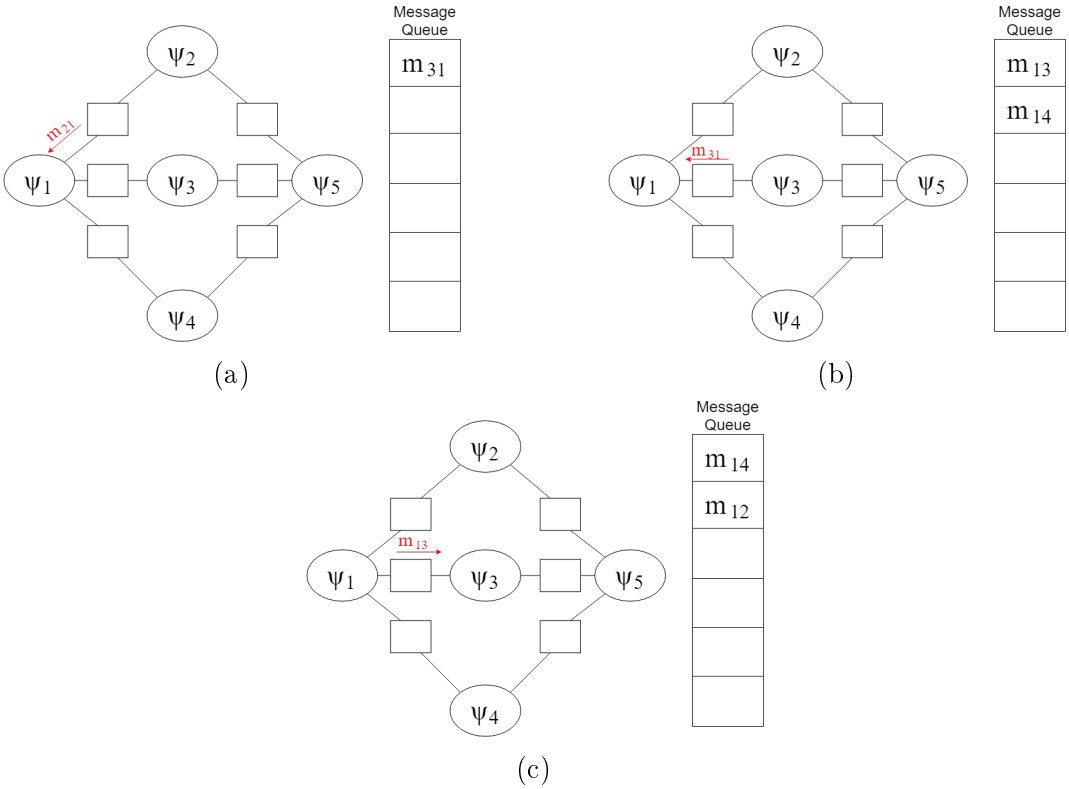


FIGURE 4.6: Two threads (thread 0 and 1) are used to do inference on this CG. (a) m_{21} is being passed. Only one thread can write to ψ_1 at a time, therefore only m_{21} can be calculated. Thread 0 adds ψ_1 to the Busy Write Set and ψ_2 to the Busy Read Set to pass m_{21} and thread 1 waits. (b) Thread 1 adds ψ_1 to the Busy Write Set and ψ_3 to the Busy Read Set and passes m_{31}. Now thread 0 has to wait for thread 1 to finish passing m_{31}. (c) Thread 0 adds ψ_1 to the Busy Read Set and ψ_3 to the busy write set and passes m_{13}. Thread 1 has to wait before popping another message.

To overcome this congestion, we remove the reliance on two clusters at a time. Simply locking the clusters one at a time (once when marginalising the source cluster and once when absorbing into the destination cluster) would still cause the threads to often wait at the same destination cluster. Instead, a different approach is taken in the next section.

4.4 Split message schedule

This section explains the sequential and parallel split message schedules (Split-MS), both contributing to the overall speed-up of inference in their own way.

This technique was originally designed to overcome the parallel congestion of high degree CGs. However, the sequential version has made a large improvement over the standard RBU algorithm, both in speed-up and messages sent before convergence.

4.4.1 Sequential split message scheduling

Split-MS changes the way that messages are added to the message queue. The calculation of one message is split into its two parts: marginalising the source cluster (to create the message); and absorbing the message into its destination cluster. These two parts get added to the message queue separately.

If the message has been calculated (by marginalising the source cluster) it adds its destination cluster onto the message queue. The destination cluster is added with a priority equal to the distance between the newly calculated message and the old message.

For the destination cluster to be calculated, it first assesses all of its adjacent sepsets for new messages. All the new messages that have been calculated on incoming sepsets are absorbed into the cluster. Thereafter, all the outgoing messages are added to the message queue with a priority equal to the priority with which the destination cluster was added to the message queue. See Figure 4.7 for an example of this. Note that the messages adjacent to the destination cluster do not use the belief residual of the cluster to determine its priority. This is due to the expense of calculating belief residuals of large clusters.

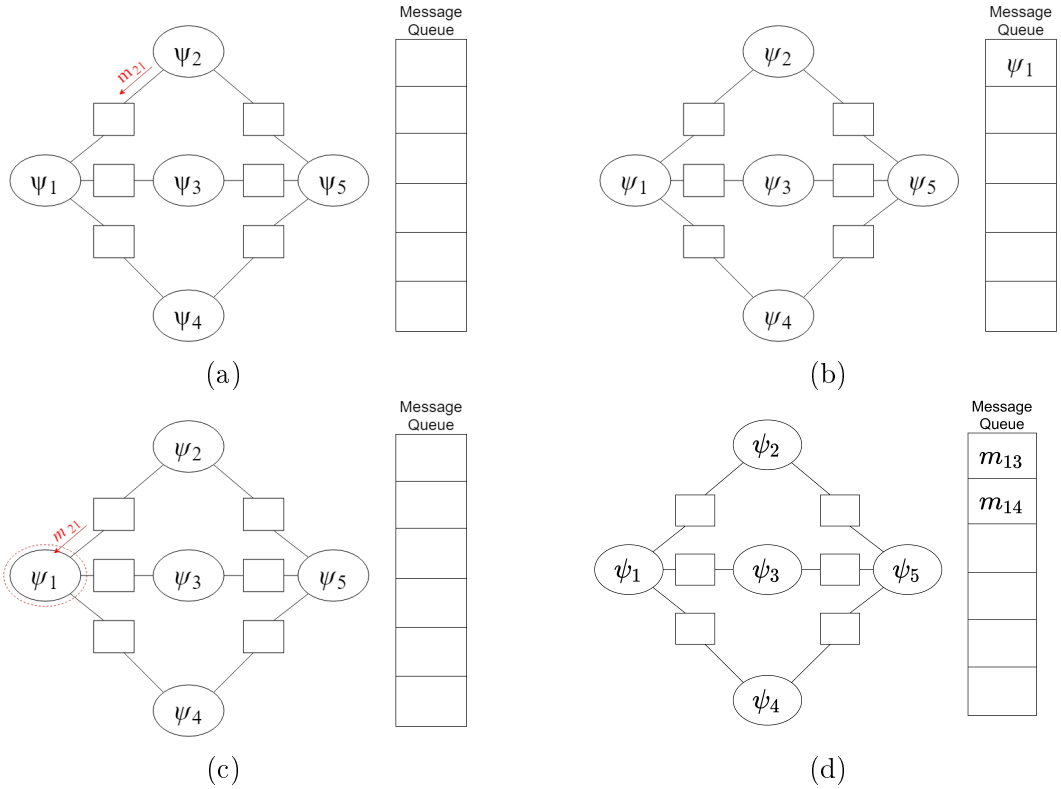


FIGURE 4.7: (a) m_{21} is being calculated. (b) ψ_1 is added to the message queue by m_{21}. (c) m_{21} is absorbed into ψ_1. (d) ψ_1 adds its adjacent messages onto the queue.

There is a possibility for two messages to be calculated along a single sepset (one in each direction) before either of them are absorbed into their destination clusters. Therefore, we double the number of sepsets and make them directional. This prevents the messages from over-writing each other.

Advantages

Sometimes a message that has already been calculated is added to the queue again before it has been absorbed by its destination cluster. When this occurs, the old message is discarded and only the new message is absorbed into the destination cluster. It would seem that this wastes processing time, due to the calculation of unnecessary messages.

However, it has the opposite effect. Messages carry more updated information. Therefore, fewer messages have to be absorbed into their destination clusters. This reduces the number of messages sent before convergence.

Multiple messages to a cluster can be calculated before the cluster is calculated. This allows the cluster to absorb more information before it adds its adjacent messages to the queue. Therefore, these newly added messages carry more information than they would otherwise, further reducing the number of messages sent before convergence.

Disadvantages

Due to the doubling of the amount of sepsets stored, more memory is used for every CG. This could cause problems for large CGs on systems with limited memory.

4.4.2 Parallel split message scheduling

Here we explain the benefit of the parallel version of the Split Message Scheduling method we proposed. In Figure 4.8 we can see how messages and clusters are calculated in parallel.

In Figure 4.8, two threads are used to do inference on this CG. m_{21} and m_{31} are being calculated by thread 0 and 1, respectively. Once they have finished, ψ_1 is re-added to the queue by both of the threads, increasing the priority of ψ_1. Thereafter, m_{41} and ψ_1 are popped from the queue and calculated by threads 0 and 1, respectively. ψ_1 is added to the queue again by thread 0. Thread 1 adds m_{12}, m_{13} and m_{14} to the queue. Finally, m_{45} and ψ_1 are popped from the queue and calculated.

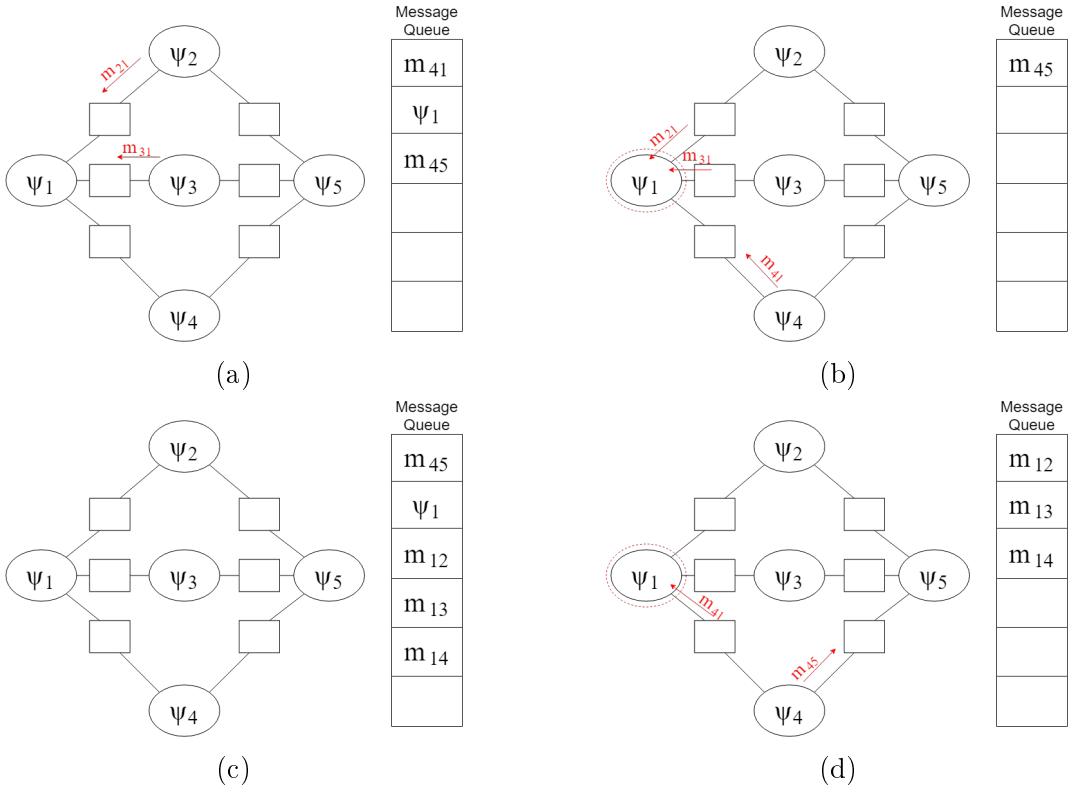


FIGURE 4.8: Two threads (threads 0 and 1) are used to do inference on this CG: (a) m_{21} and m_{31} are being calculated by threads 0 and 1, respectively. m_{41}, ψ_1 and m_{45} are in the queue. (b) m_{41} and ψ_1 are popped from the queue and calculated; (c) m_{12}, m_{13} and m_{14} are added to the queue and ψ_1 is added to the queue again by m_{21} and m_{31}; (d) m_{45} and ψ_1 are popped from the queue and calculated.

Advantages

As is clear from Figure 4.8, the parallel Split-MS further reduces the waiting time of threads, as there are fewer clashes over locked clusters.

Disadvantages

Shared memory issues At this point, the shared message queue prevents any further speed-up, as it sequentialises the process.

Consider the two threads in Figure 4.9 (threads 0 and 1) doing inference on a CG. Thread 0 locks access to the message queue, pops a message and releases the lock on the message queue. Thereafter, it passes the message. Thread 1 waits until thread 0 has released the lock before it can access the message queue. Thread 1 then locks the message queue, pops a message and releases the lock. Finally, it can start passing its message. This process repeats. Depending on the time it takes for each message to be passed, the time each thread spends waiting can differ.

From this example we can see how access to the message queue can cause parts of the code to run sequentially. It is also intuitive that with more threads, the threads are waiting to access the message queue more often and for a longer time.

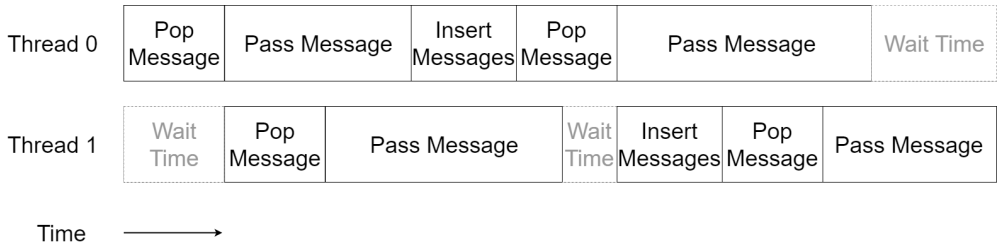


FIGURE 4.9: This figure illustrates the waiting time caused by shared message queues. Threads have to take turns to pop from or insert into the message queue.

A solution for this issue is suggested in Section 4.4.3.

4.4.3 Parallel split message scheduling with manager thread

This method builds on parallel Split-MS by adding a manager thread. The job of the manager thread is to take sole control of the message queue.

The manager thread pops messages from the queue and hands it to the worker threads. When the worker threads are done calculating a message or absorbing messages into a cluster, they signal the manager thread. The manager thread receives the messages or clusters that each worker thread wants to add to the queue and adds it for them. This allows the worker threads to only work on the calculations and reduces their overhead.

The following steps correspond to that of Figure 4.10:

1. The manager thread (thread 0) attempts to pop m_{25}. Therefore, it has to check if the busy write set contains ψ_2. With ψ_2 already in the busy write set, m_{25} is skipped.

2. The manager thread attempts to pop ψ_1. ψ_1 is in neither the busy write or read sets. It is subsequently added to the busy write set.

3. ψ_1 is added to the inbox of worker thread 1.

4. The manager thread attempts to pop m_{35}. m_{35} is not in the busy write set. It is subsequently added to the busy read set. Meanwhile, worker thread 1 fetches ψ_1 from the inbox and starts absorbing the relevant adjacent messages.

5. m_{35} is added to the inbox of worker thread 1. Meanwhile, worker thread 2 finishes its calculations. It proceeds to add the messages adjacent to ψ_2 into its outbox.

6. The manager thread attempts to pop m_{25}. Therefore, it has to check if the busy write set contains ψ_2. With ψ_2 already in the busy write set, m_{25} is skipped.

7. The manager thread removes ψ_2 from the busy write set, pops the messages from the outbox of worker thread 2 and inserts them into the message queue. Worker thread 2 fetches m_{35} from its inbox and starts calculating the message. The manager

thread attempts to pop m_{25} and succeeds, because ψ_2 is no longer in the busy write set. It is subsequently added to the busy read set.

8. m_{25} is added to the inbox of Worker Thread 1.

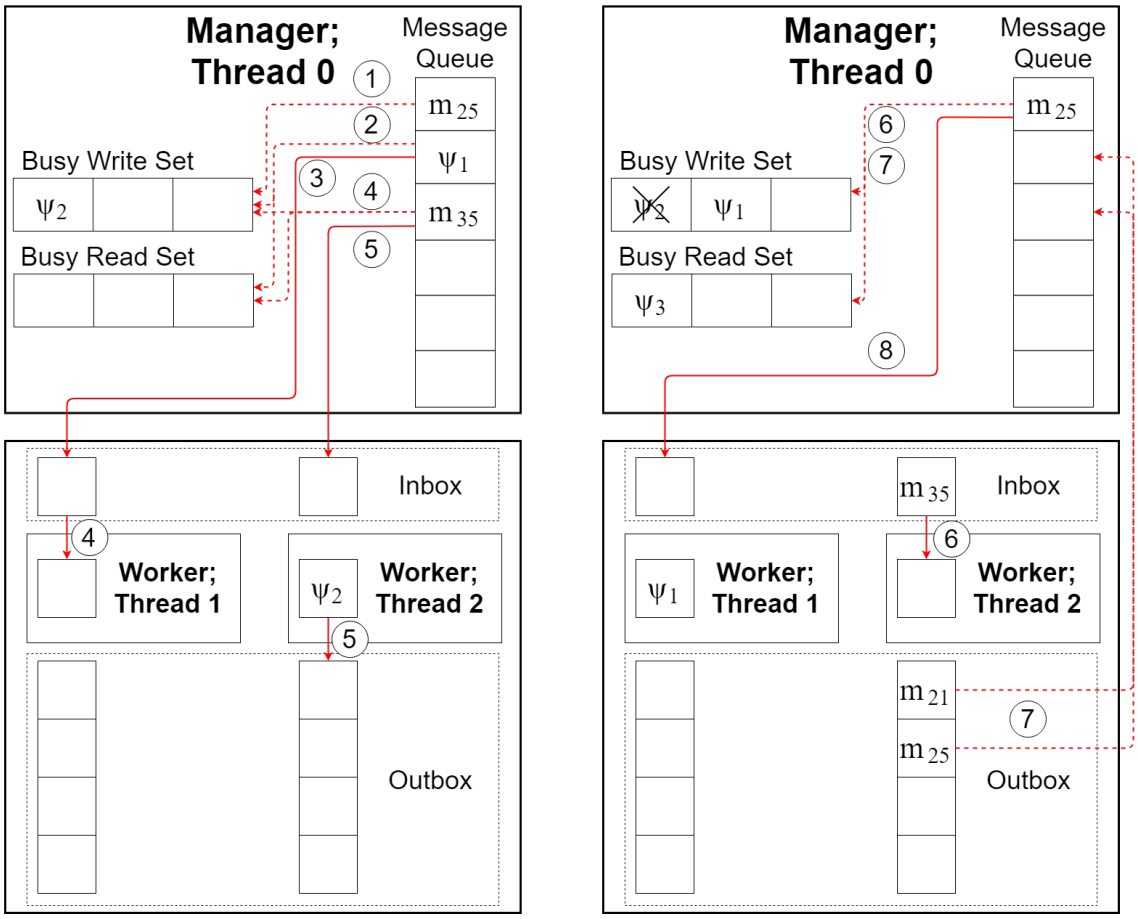


FIGURE 4.10: (a) The manager thread pops items from the message queue to add to the inboxes of the worker threads. Worker thread 2 finishes calculating a cluster and adds its output to its outbox. As soon as worker thread 1 has ψ_1 available in its inbox it fetches it and starts calculating it. (b) The manager thread has to remove ψ_2 from the busy write set and empty the outbox of worker thread 2 before it can pop m_{25} from the queue.

Advantages

The use of a manager thread allows the worker threads to focus on the calculation of messages. The worker threads have virtually none of the management overhead.

Neither the message queue nor the busy sets are shared. Therefore, no locks are required on them. This reduces the idle time of threads.

Disadvantages

Sacrificing one thread as a manager thread limits the theoretical max speed-up of the parallelisable part of the code to $N-1$, where N is the number of threads available. This is compared to the theoretical max speed-up of N for the normal parallel Split-MS.

At a certain number of threads, the manager thread's workload becomes too much. The worker threads start waiting for the manager thread to assign them work. The manager thread becomes a hard bottleneck.

4.4.4 Multiple manager threads

To expand upon the limitations of having a single manager thread to do Split-MS, we experimented with having more than one manager thread.

In Figure 4.11, an example with two manager threads and two worker threads is shown. Each of the manager threads has its own message queue. The manager threads share the busy write and read sets.

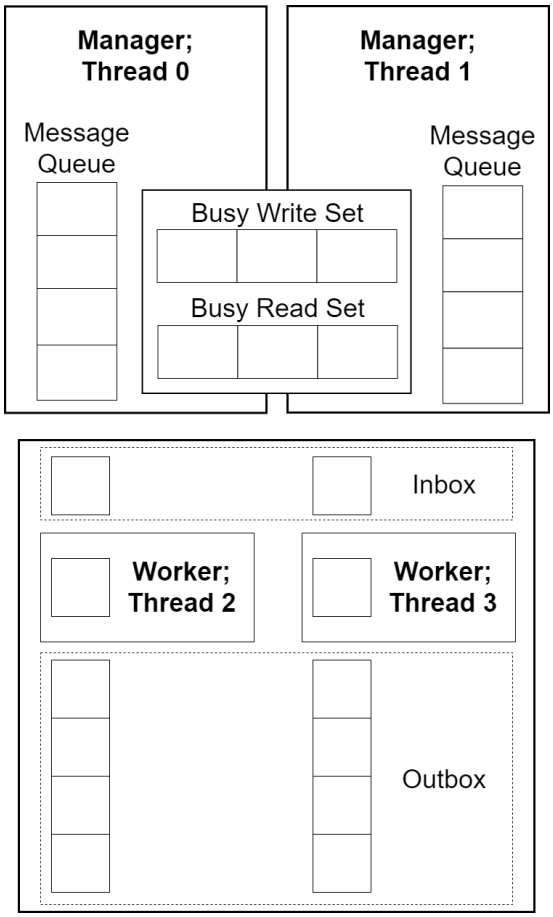


FIGURE 4.11: There are two manager threads in this setup. Each manager thread has their own message queue to prevent a shared memory clash. The busy sets have to remain global to prevent any clusters from being used simultaneously by more than one worker thread. The worker threads remain the same as in the single manager thread example.

The manager threads both add items from the message queues to the inboxes of all worker threads. They also fetch new items from the outboxes of all worker threads. The decision as to which manager thread does any work is random. Both of the manager threads loop over all the worker threads. The first manager thread to get to a worker thread that has an empty inbox or a non-empty outbox is the one who does the work.

A lock is required on the busy write and read sets. This is due to these sets being shared by the manager threads.

Advantages

This technique should allow for the expansion of the algorithm to processors with more threads.

Disadvantages

The message queue that is split between the number of manager threads could theoretically increase the number of messages sent before convergence. This is due to some messages of a lower priority being at the top of one message queue while the other message queue contains all the higher priority messages. They are popped from their respective queues with equal importance.

The busy sets are shared. This means that a lock is required on each set. The locks cause the manager threads to wait for each other before they can add or remove items from the busy sets. This could cause the extra manager threads to be impractical.

4.5 Summary

This chapter explained the different parallel message passing schedules that were created for this work. It was shown how the algorithms explained in this chapter build on the sequential residual belief update schedule. The advantages and disadvantages of each algorithm were discussed to determine what could lead to the best possible speed-up.

5 Evaluation tasks and methodology

This chapter explains what was measured to test the algorithms created in Chapter 4. It explains which cluster graphs (CGs) the algorithms were tested on. It also shows the environment in which the tests were run. All the test results are shown in Chapter 6.

5.1 Cluster graphs used for evaluation purposes

The PGM representation used for all the tests is the cluster graph (CG). For a detailed look at the structure of the specific CGs, see Appendix B. It illustrates the exact sudoku CGs that were used. Unfortunately, the satellite image denoising CG was too big to generate an image of it. The following is the specifications of each of the CGs that were tested on:

- Sudoku solver with a cluster size of seven RVs (each with nine possible values):
  - Number of clusters: 76 clusters.
  - Number of sepsets: 244 sepsets.
  - Size of biggest cluster: 181440 combinations (with non-zero probabilities).
- Sudoku solver with a cluster size of eight RVs (each with nine possible values):
  - Number of clusters: 35 clusters.
  - Number of sepsets: 87 sepsets.
  - Size of biggest cluster: 362880 combinations (with non-zero probabilities).

- Satellite Image Denoising:
  - Number of clusters: 82371 clusters.
  - Number of sepsets: 143716 sepsets.
  - Size of biggest cluster: 578 combinations (with non-zero probabilities).

The sudoku and satellite image denoising tasks were chosen to show how well the algorithms do on both small and large CGs. The sudoku puzzle also has more interconnections than the satellite image denoising task does. This causes the occurrence of more thread waiting time at busy clusters.

5.2 Measurements

The following measurements were recorded: time taken to converge, from which the speed-up is calculated; number of messages sent before convergence; and the accuracy of the converged result.

Speed-up is the most important measurement, given that the accuracy of the result remains intact. The number of messages sent before convergence has a large effect on the time taken to converge. It was measured to see whether this influenced each of the timings.

All the algorithms were compared to the benchmark. The benchmark is the sequential residual belief update (RBU) algorithm and its natural parallelisation. The natural parallelisation of RBU is pull message scheduling (Pull-MS).

5.3 Environment of tests

The computer used for the tests has the following specifications:

- INTEL I7 8700 3.2GHZ 6C 12T 9MB LGA1151 processor (6 cores and 12 threads)
- MSI Z370 GAMING+ LGA1151 4*DDR4, M.2, DP motherboard
- 2 x ADATA 8GB DDR4 2400(PC4-19200) DIMM CL17 RAM (16GB in total)
- The operating system (OS) used is Ubuntu 16.04

Bash commands were called from Jupyter Notebook to compile and run the different algorithms. No other programs were open during the tests.

There were always processes in the background, run by the OS. Therefore, no two runs were the same and there was always interference from the OS. For this reason, the scheduling of the threads and the order in which the messages were sent was never the same. This lead to different convergence for every run. For this reason, an average over many runs was required to get an accurate result of the time taken to converge and the number of messages sent before convergence. Every test was run 10 times. This gave a reasonable average for us to analyse all the algorithms. More runs would have given more accurate results, but due to time constraints it was not feasible to run any more tests.

5.4 Summary

This chapter explained what was measured to test the algorithms created in Chapter 4. It discussed the CGs the algorithms were tested on. This chapter also discussed the environment in which the tests were run to better understand the constraints of the tests.

6 Experiments and results

This chapter shows the results of all the algorithms that were shown in Chapter 4. These algorithms were: single push message scheduling (SPush-MS); multiple push message scheduling (MPush-MS); pull message scheduling (Pull-MS); smart message scheduling (Smart-MS); split message scheduling (Split-MS); and Split-MS with manager threads.

These experiments were created as explained in Chapter 5. The results show the speed-up of each algorithm on different sizes of CGs. The results also show the number of messages sent before convergence.

6.1 Satellite image denoising experiments

For the image denoising task in Figures 6.1(a) and 6.2(a) it is clear that Split-MS with one manager thread has the best results. However, there is little speed-up after six threads for any of the algorithms. This indicates that all of these algorithms reach a point where the sequential parts of the algorithms prevent any further speed-up.

Specifically, for all algorithms, except Split-MS with manager threads, the cause of the plateau at six threads is caused by the shared message queue. The message queue causes the algorithms to have a large sequential part, due to the threads having to access the message queue one at a time.

For the Split-MS with one manager thread, a plateau also starts forming at six threads. This is due to the load on the manager thread becoming too large. The worker threads have to wait after finishing with one message before they receive a new message because the manager thread is working too slowly.

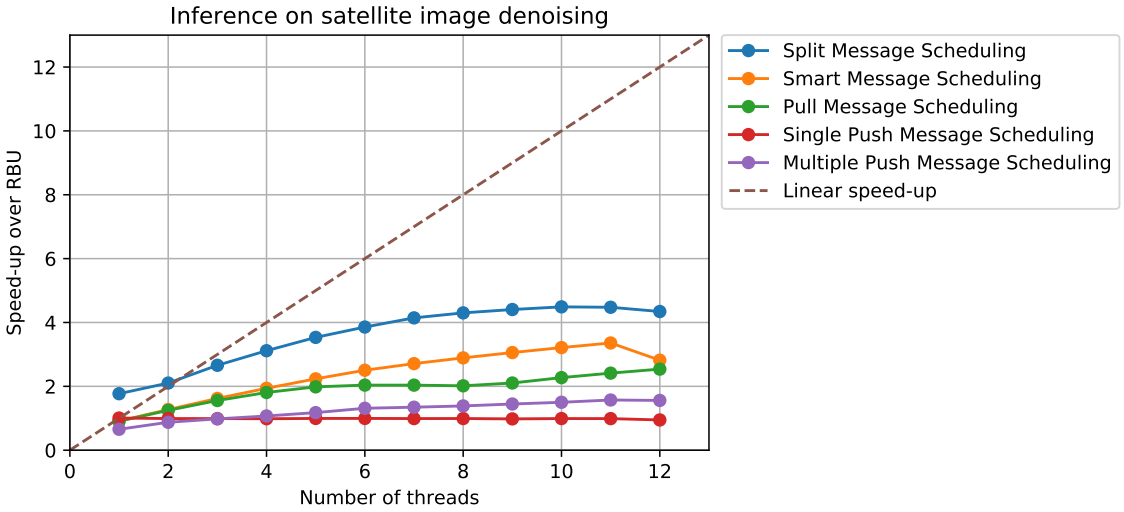
For Split-MS with two and three manager threads speeding up still occurs after six threads. However, due to the loss of more worker threads (that become manager threads), Split-MS with two and three manager threads is slower than both normal Split-MS and Split-MS with one manager thread. It is possible that, given more threads, Split-MS with two and three manager threads could outperform the other algorithms. Experiments would have to be run on a processor with more threads to confirm this.

Using SPush-MS does not result in a speed-up over the normal residual belief update (RBU) algorithm. This is partly due to the restrictive nature of its design. The potential speed-up is limited to the degree of connections between every cluster (as explained in Section 4.1.1). In the case of the satellite image denoising task, there is at most a four degree connection between clusters. This means that at most three messages are sent at the same time. Therefore, there is no speed-up after three threads. The fact that there is no speed-up from one to three threads is due to the overhead of parallelising and the time threads spend waiting for other threads when one message is calculated faster than another (as explained in Section 4.1.1).

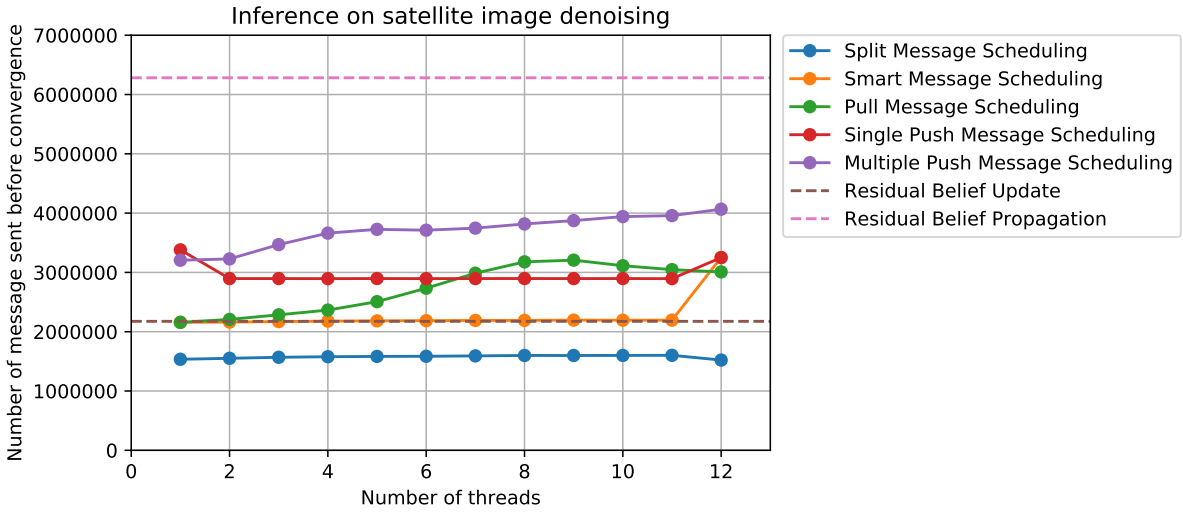
The speed-up of all algorithms, except SPush-MS, follows a similar trajectory. However, the reason for the speed-up difference is due to varying levels of sequential code (caused by shared memory access) and parallel overhead. This is indicative of Amdahl's law (explained in Section 3.2.4). Therefore, it is clear that the Split-MS technique has the least

parallel overhead and the least sequential code.

In Figures 6.1(a) and 6.2(a) the satellite image denoising task results are shown. The best speed-up for this task is in Figure 6.2(a), by Split-MS with one manager thread. The speed-up starts to plato at 10 threads, with a speed-up of 5.3 times.



(a)



(b)

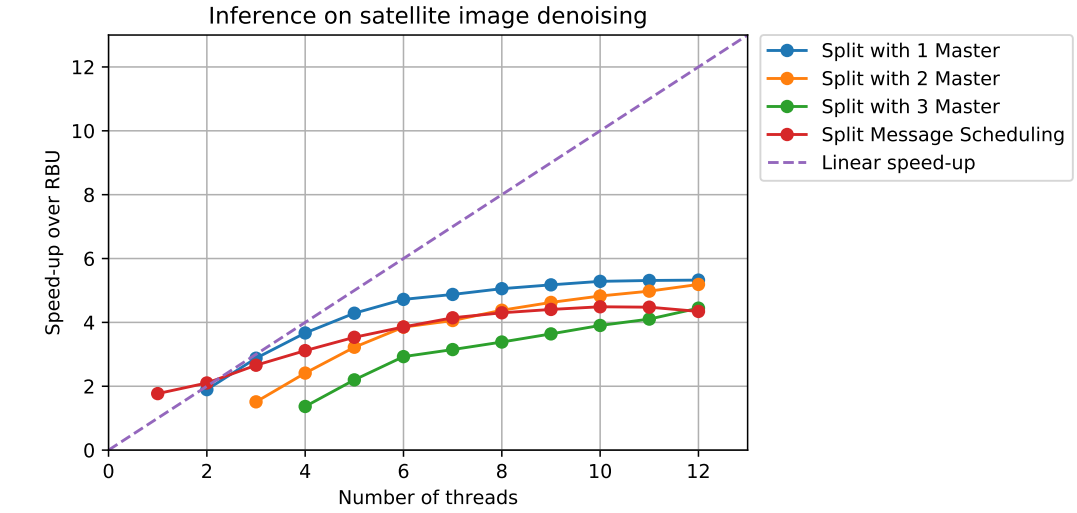
FIGURE 6.1: (a) The speed-up of all the designed algorithms are compared. (b) The number of messages sent before convergence for each of these algorithms is shown.

Consider Figures 6.1(b) and 6.2(b). They show the number of messages sent before convergence of all algorithms for the satellite image denoising task. Note how the push algorithms are outperformed by the pull algorithms. Remember that all the algorithms (even RBU), except for SPush-MS and MPush-MS, use the pull message passing technique. This proves that the pull method leads to better convergence than the push method.

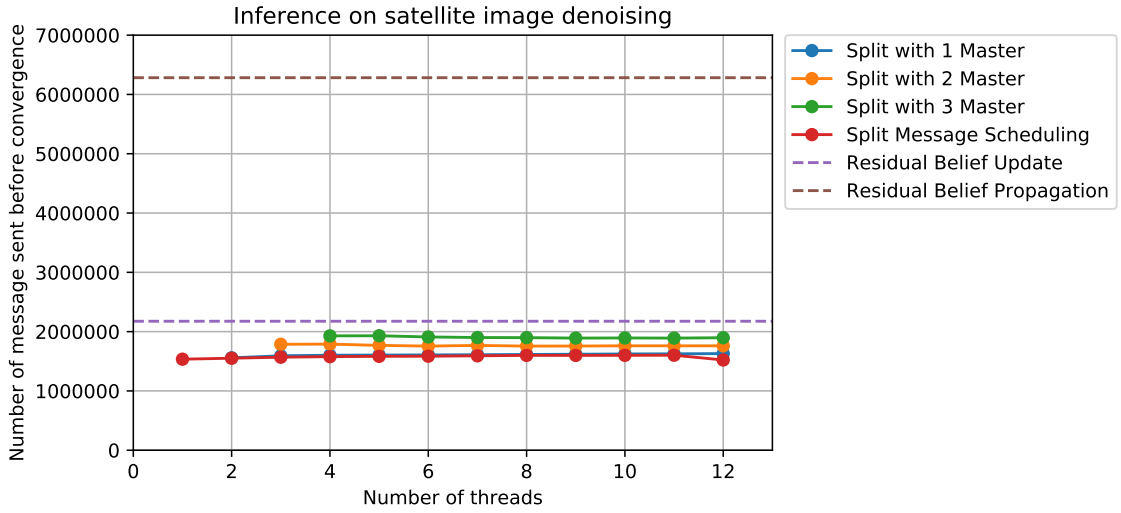
The Split-MS variants outperform the RBU algorithm and its natural parallelisation (Pull-MS) by converging in fewer messages. Note how Split-MS and Split-MS with one manager thread have almost identical convergence rates. They have very similar message passing orders. The reason Split-MS with two or three manager threads sends more messages before convergence is due to the message queue being split up between the manager threads. One message queue might contain more of the important messages, while the other contains only less important messages. This imbalance causes less important messages to be calculated before the important messages are calculated, leading to slower convergence.

The split message queue in Split-MS with two or three manager threads also allows duplicate messages to exist in the different message queues. Normally, if a message gets added to a message queue, a check is done to see whether the message is already in the message queue. Doing this check over multiple message queues is very expensive. Therefore, the check is only done on the message queue it is to be added to, allowing duplicates to exist in the different message queues. These duplicate messages are then calculated unnecessarily.

Note that, in general, the messages sent before convergence are only slightly affected by the number of threads used. Only Pull-MS and MPush-MS are greatly affected by an increasing number of threads.



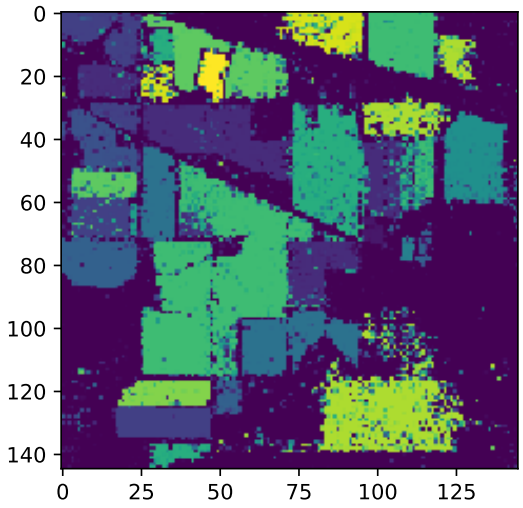
(a)



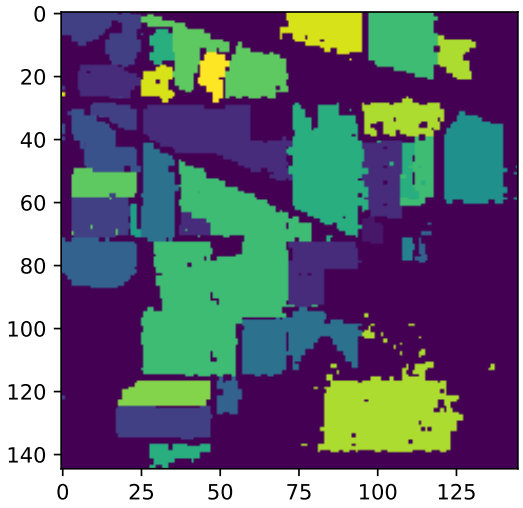
(b)

FIGURE 6.2: (a) The speed-up of Split-MS and its variants are compared. (b) The number of messages sent before convergence for each of these variants is shown.

In Figure 6.3(a) we see the noisy satellite image before being denoised by all the algorithms. Figure 6.3(b) shows a typical denoised version of the satellite image. All the algorithms converged to a similar result. Therefore, only one of them is shown. For a full illustration of how each algorithm denoised the image, see Appendix A.



(a)



(b)

FIGURE 6.3: (a) The noisy satellite image to be denoised. (b) A typical denoised version of the satellite image created by one of the inference algorithms.

6.1.1 Further analysis of Split-MS

To further analyse the performance of Split-MS, we show some low-level inspection, as well as the percentage of time spent in certain parts of the code.

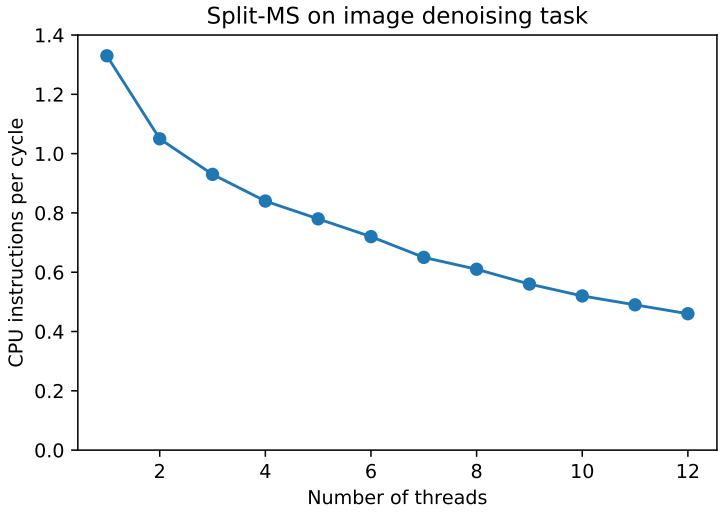
In Figure 6.4, the instructions per core cycle for Split-MS and Split-MS with one manager thread are shown. The more threads are used to do inference, the worse the results become. This is due to the nature of the architecture of the CPU on which the tests were run (the system specifications are shown in Section 5.3). Some possible causes for the decline in instructions per core cycle are discussed below.

The cause of this decline in performance is not due to cache misses. In fact, cache misses become fewer with more threads during the execution of Split-MS on the satellite image denoising task. This is displayed in Figure 6.5, which shows the cache misses for Split-MS.

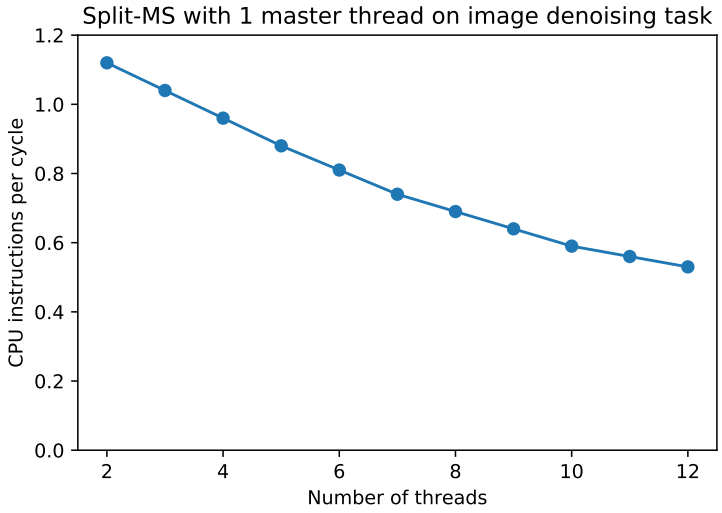
The hyper-threading (explained in Section 3.2.4) on the processor is a possible reason for this decrease, because there are two threads on one core. These threads share execution resources. Therefore, an N times speed-up with N threads is not possible.

As explained in 3.2.4, hyper-threading can also cause a bottleneck at the memory bandwidth, because there are more threads that have to use the shared memory bandwidth.

Another cause of the low speed-up is the CPU migrations occurring during the execution of the process. Figure 6.6 illustrates this on Split-MS. These CPU migrations waste valuable execution time.



(a)



(b)

FIGURE 6.4: (a) The CPU instructions per cycle for Split-MS on the image denoising task; (b) The CPU instructions per cycle for Split-MS with one manager thread on the image denoising task.

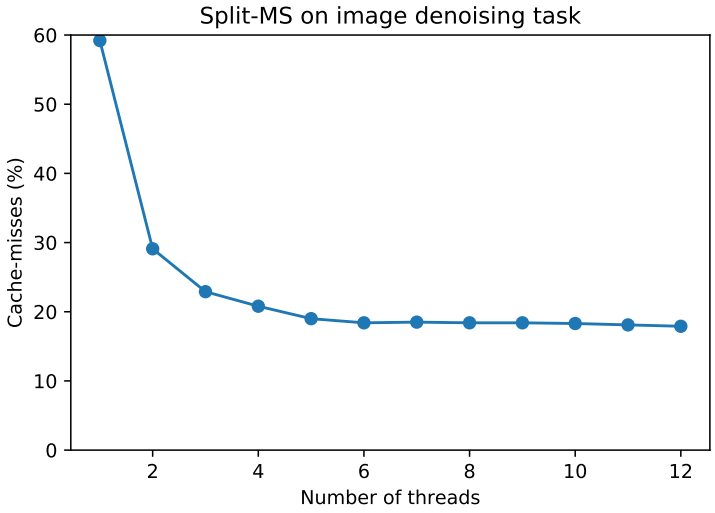


FIGURE 6.5: The percentage of cache-misses decreases as the number of threads used increases.

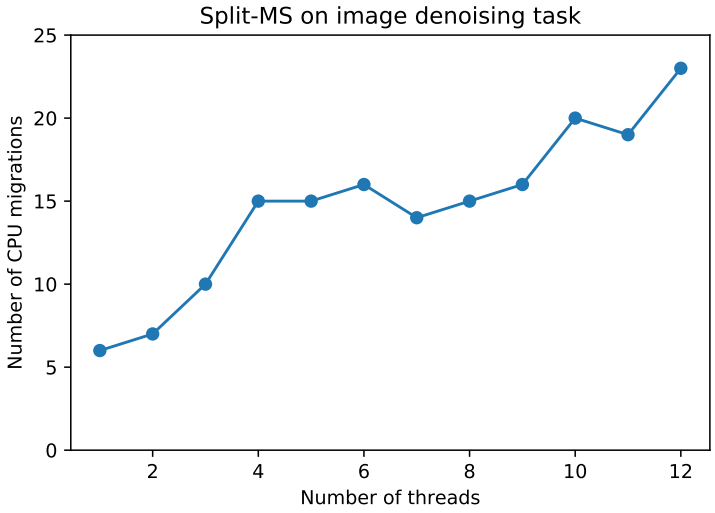


FIGURE 6.6: The number of CPU migrations increases as the number of threads used increases.

In Table 6.1, the average percentage of time it took for threads to wait at each lock that is used is shown. This was measured using 12 threads. It is clear that the most time is lost while waiting for access to the message queue. For this reason, manager threads were added to Split-MS.

Lock	Percentage of total time
Message lock	38.31 %
Old messages lock	1.70 %
Busy set lock	0.34 %
Sepset beliefs lock	2.38 %

TABLE 6.1: The average time threads wait at each lock used in Split-MS on the satellite image denoising task, relative to the total time of execution.

In Table 6.2, the number of calculations of Split-MS on the image denoising task is shown. This includes the number of messages that were calculated, the number of clusters that were updated and the number of messages that were absorbed into clusters.

Some messages do not get absorbed. These waste processing time. The percentage of wasted messages was 38%. However, this was the advantage of Split-MS. Some outdated messages were overwritten before they got absorbed. A more up-to-date message was absorbed instead, reducing the number of messages sent before convergence. Some clusters also absorbed more than one message at a time. Thus, when these clusters finished updating, the adjacent messages that were added to the message queue contained more information than in other algorithms.

Type of calculation	Number of calculations
Total cluster calculations:	394779
Total message calculations:	1080839
Total messages in cluster calculations:	782691

TABLE 6.2: The number of clusters that was updated, the number of messages that was calculated and the number of messages absorbed into clusters for Split-MS on the satellite image denoising task are shown.

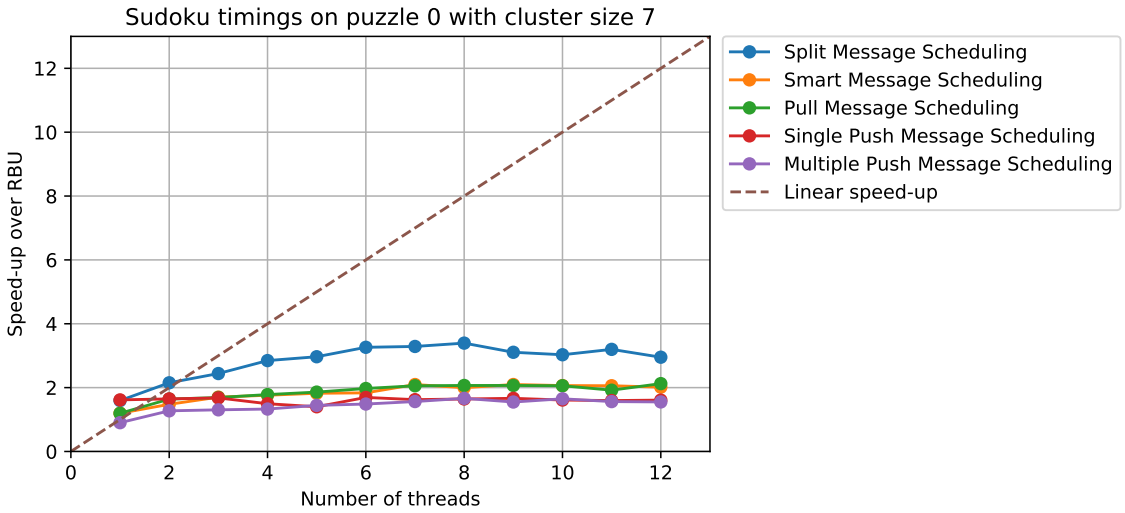
For Split-MS with manager threads, the message queue is only accessed by the manager thread. As discussed before, at a higher thread count the manager thread becomes the bottleneck. For Split-MS with one manager thread, running the image denoising task, 80.50% of the manager thread's time is spent accessing the message queue. The rest of the time is spent on loading the inbox, unloading the outbox, overhead and managing the busy sets. The only way to gain further speed-up is by reducing the number of accesses to the message queue or by reducing the time it takes to insert and pop from the message queue. Unfortunately, the insert and pop from the queue are already optimised, so no further speed-up can be gained here.

6.2 Sudoku puzzle experiments

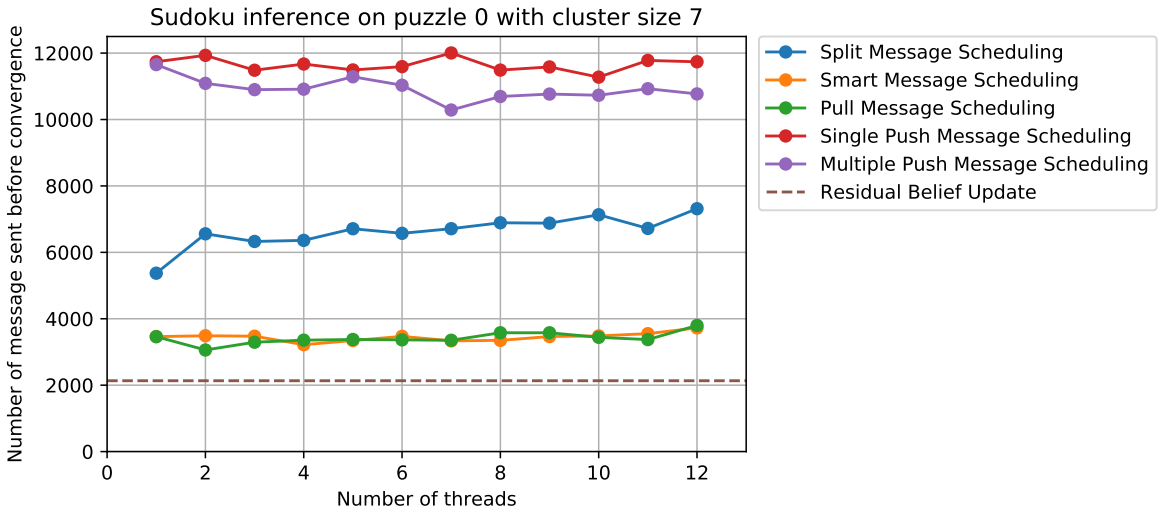
Comparing the performance of cluster sizes in Figures 6.7(a), 6.8(a), 6.9(a), 6.10(a), 6.11(a), 6.12(a), 6.13(a) and 6.14(a) it is clear that the smaller a CG is the less speed-up is gained from parallelising. This is caused by the increased number of memory clashes where threads attempt to write to the same cluster, due to the size of the CG. Even for Split-MS, there comes a point where no more clusters are available to be given to a thread. Some threads have to wait for others to complete their message calculation before they can be given a new message.

For both the puzzles used in this experiment similar results were seen for the same cluster sizes, further suggesting that the size of a CG has an effect on the speed-up of the inference. The similarities between the two puzzles' results also suggest that this is a general rule.

The best speed-up seen in Figures 6.7(a) and 6.8(a) is by Split-MS. The speed-up plateaus at eight threads, with a speed-up of 3.4 times.



(a)



(b)

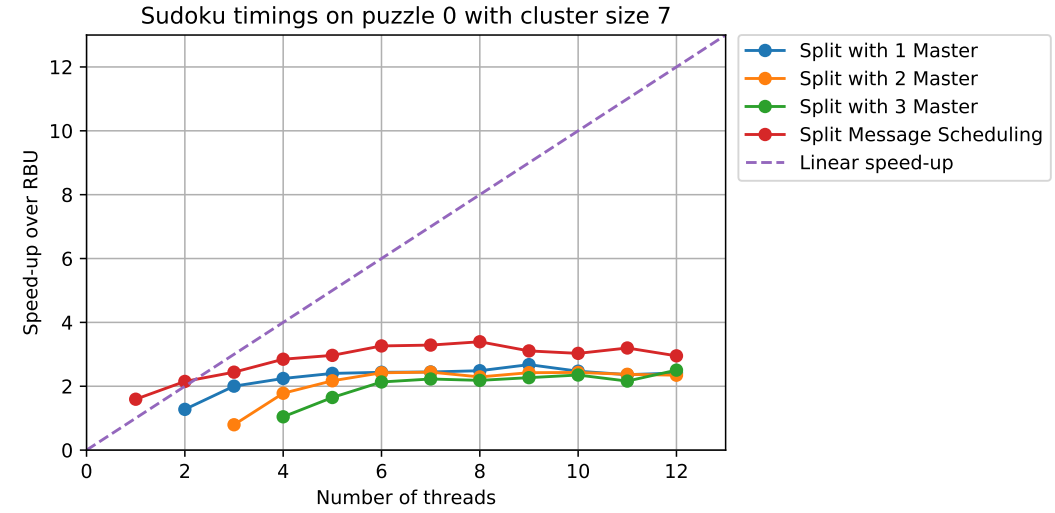
FIGURE 6.7: The speed-up relative to RBU and the number of messages sent before convergence for all the designed algorithms are shown for the sudoku solver using a cluster size of 7 RVs on puzzle 0.

All the puzzle and CG combinations in Figures 6.7(b), 6.8(b), 6.9(b), 6.10(b), 6.11(b), 6.12(b), 6.13(b) and 6.14(b) show that the number of messages sent before convergence for the sudoku application is worse than the benchmark, RBU. This is another reason the

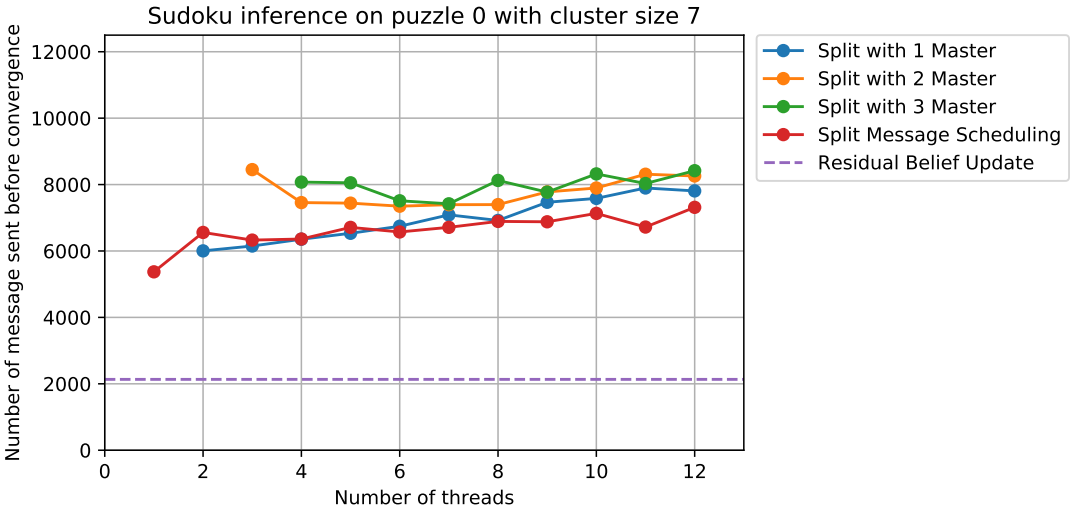
speed-ups for the sudoku application are less than that of the satellite image denoising task.

These graphs also show that pull message scheduling leads to fewer messages sent before convergence compared to push message scheduling, as explained in 2.4.

Note that there is no RBP shown in the sudoku experiments. This is due to RBP not converging for any of the puzzles or cluster sizes.



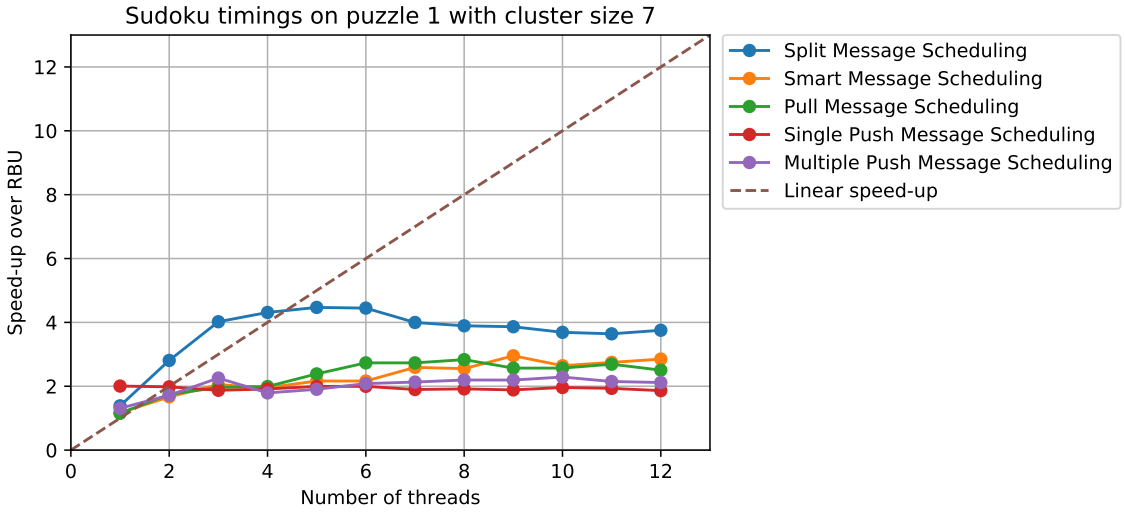
(a)



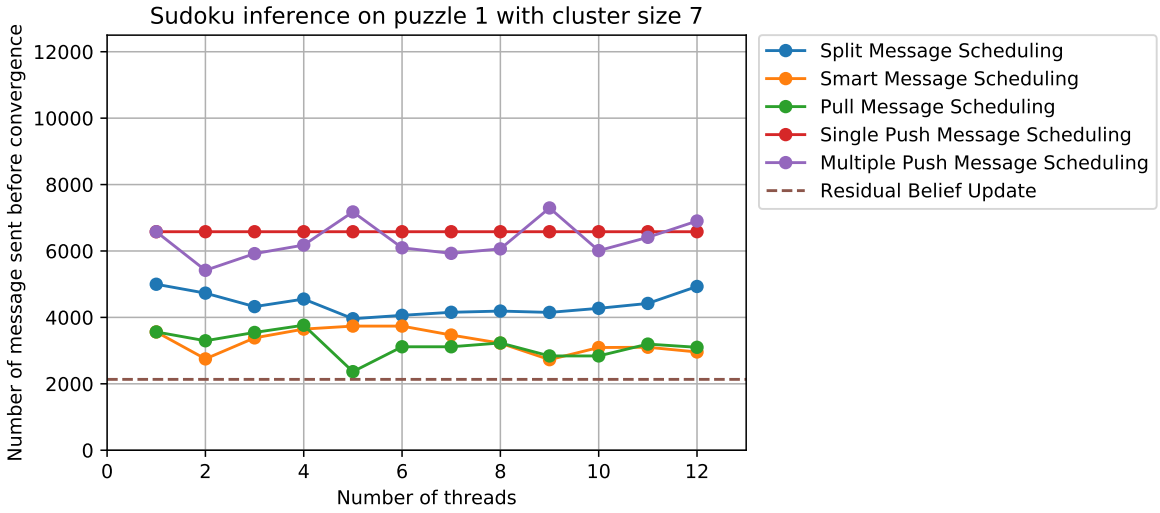
(b)

FIGURE 6.8: The speed-up relative to RBU and the number of messages sent before convergence for the Split-MS variants are shown for the sudoku solver using a cluster size of seven RVs on puzzle zero.

The best speed-up seen in Figures 6.9(a) and 6.10(a) is by Split-MS. The speed-up plateaus at six threads, with a speed-up of 4.0 times and seems to get slower after that.

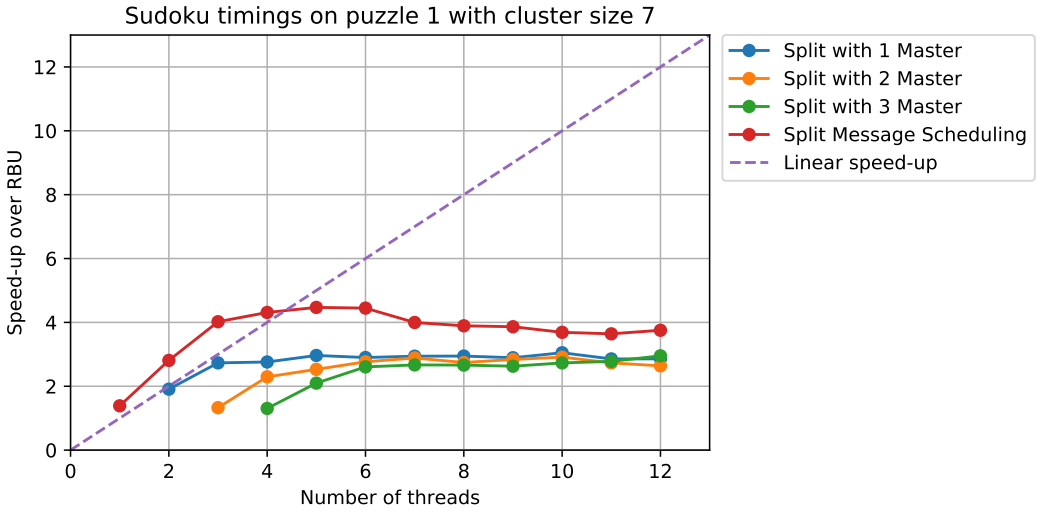


(a)

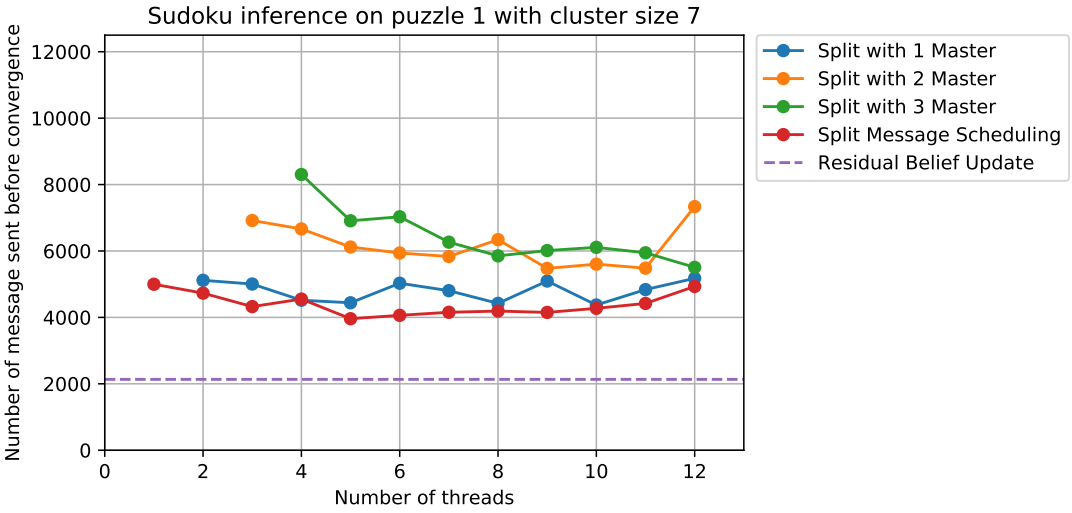


(b)

FIGURE 6.9: The speed-up relative to RBU and the number of messages sent before convergence for all the designed algorithms are shown for the sudoku solver using a cluster size of 7 RVs on puzzle 1.



(a)



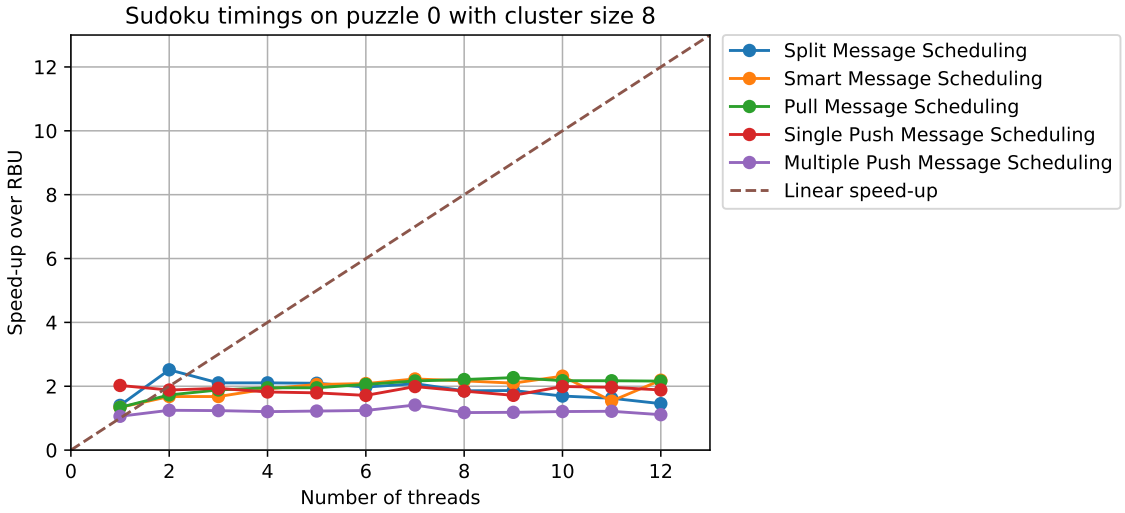
(b)

FIGURE 6.10: The speed-up relative to RBU and the number of messages sent before convergence for the Split-MS variants are shown for the sudoku solver using a cluster size of 7 RVs on puzzle 1.

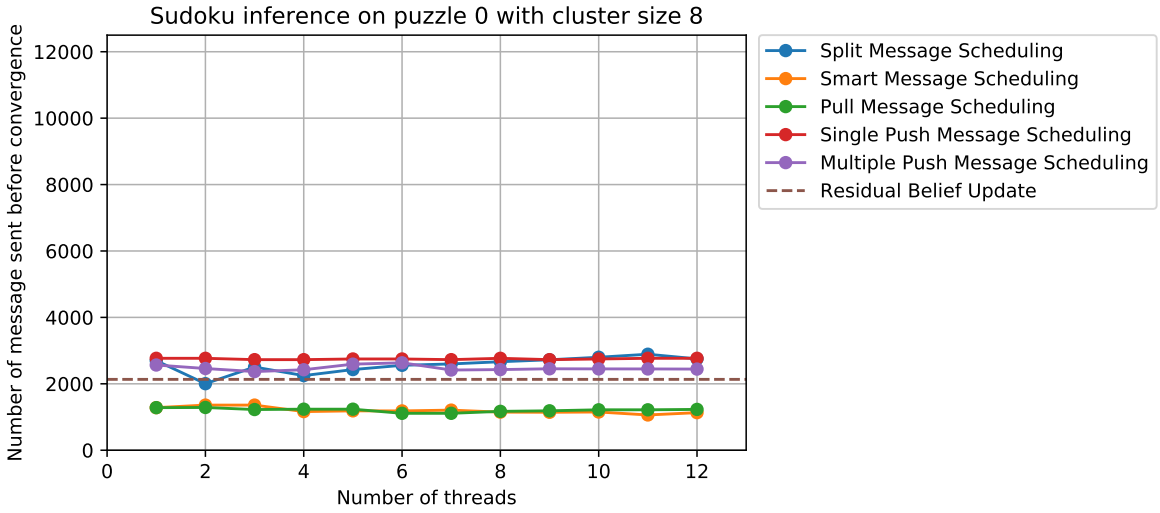
Consider Figures 6.11(a), 6.12(a), 6.13(a) and 6.14(a). The results for cluster size eight on both puzzles are erratic. This is evidence of the effect that parallelising has on the convergence behaviour. Due to the order of message calculations being different for every run, the convergence can happen faster or slower. The smaller the CG, the more prominent this effect is.

The best speed-up seen in Figures 6.11(a) and 6.12(a) is for Split-MS. At two threads the speed-up is 2.7 times.

In Figures 6.11(b), 6.12(b), 6.13(b) and 6.14(b) it is interesting to see the messages sent before convergence for the smart and pull message scheduling techniques. Both of these outperform RBU. It is as if the most important messages are not always on top of the message queue and that parallelising this specific application allows us to follow a MP schedule that consistently reaches convergence faster.

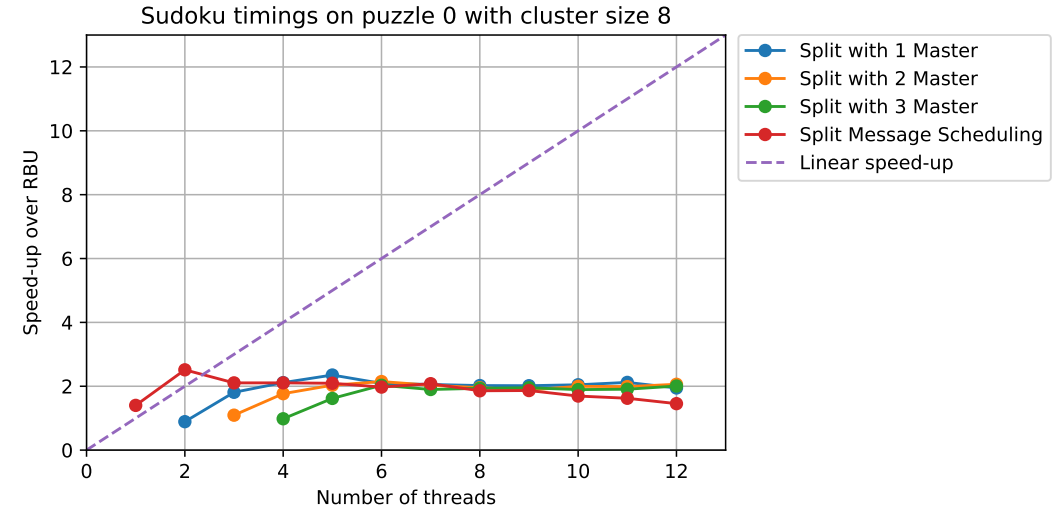


(a)

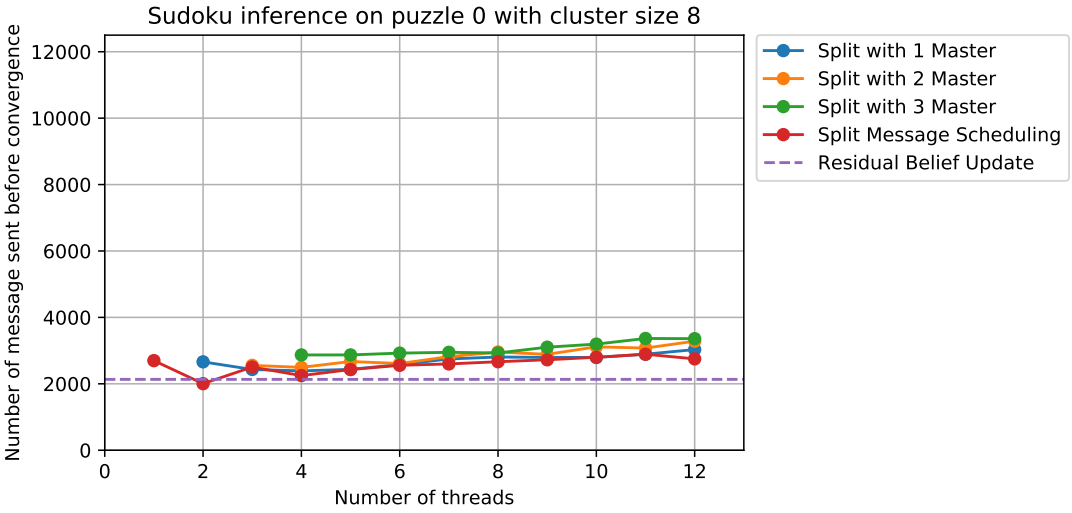


(b)

FIGURE 6.11: The speed-up relative to RBU and the number of messages sent before convergence for all the designed algorithms are shown for the sudoku solver using a cluster size of 8 RVs on puzzle 0.



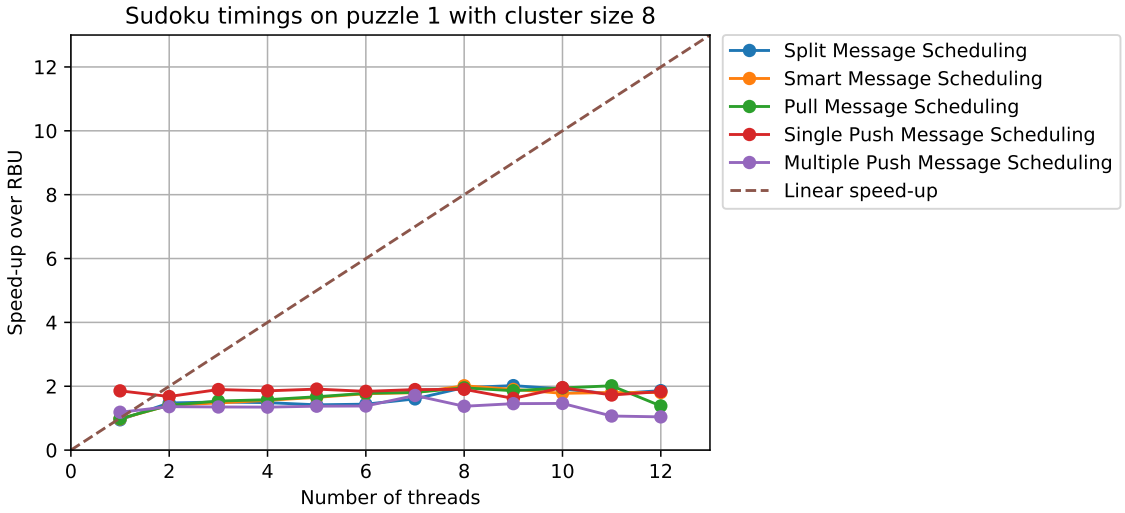
(a)



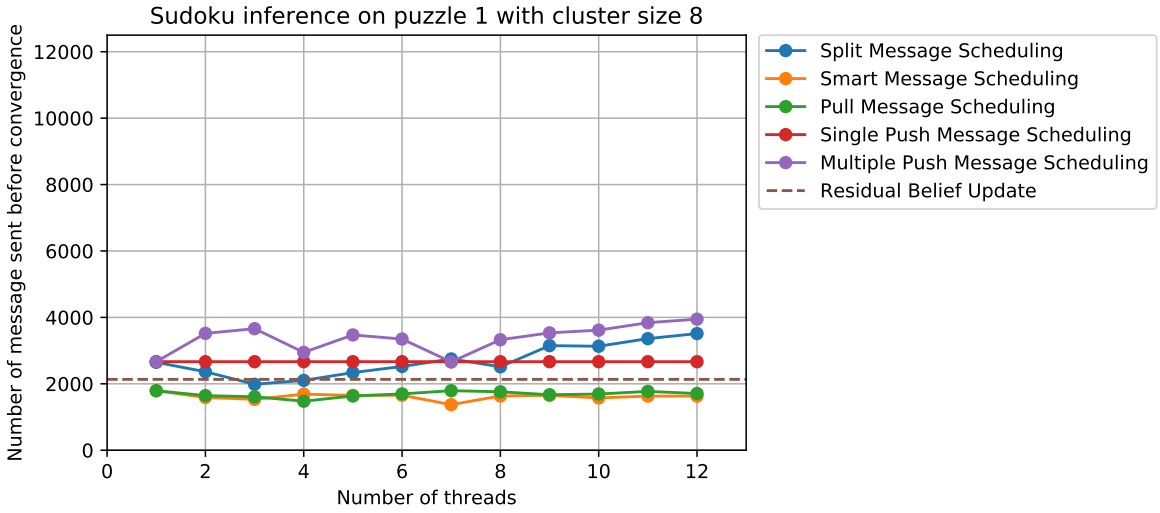
(b)

FIGURE 6.12: The speed-up relative to RBU and the number of messages sent before convergence for the Split-MS variants are shown for the sudoku solver using a cluster size of 8 RVs on puzzle 0.

Due to its erratic nature, the best speed-up seen in Figures 6.13(a) and 6.14(a) is for Split-MS with one manager thread. At 10 threads the speed-up is 2.0 times with a plateau at 5 threads.

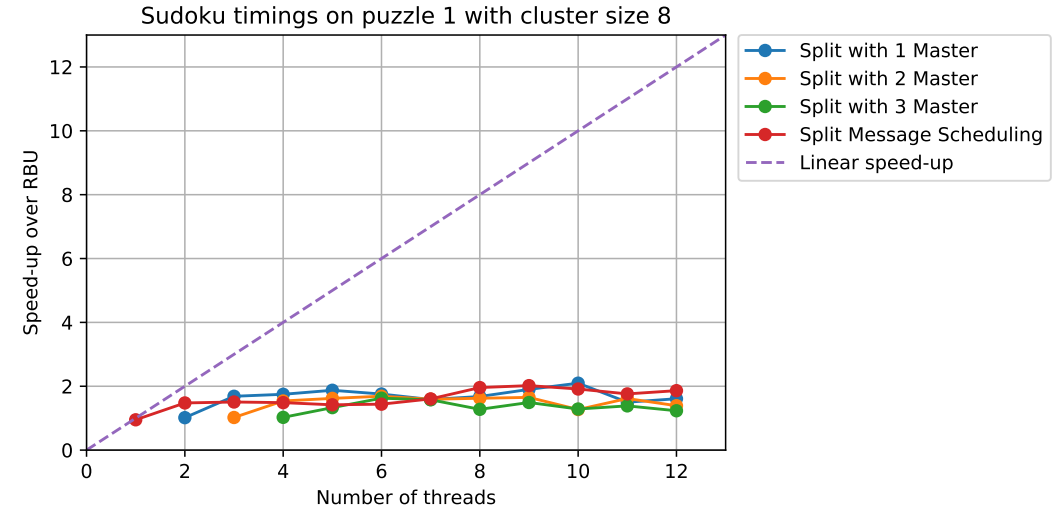


(a)

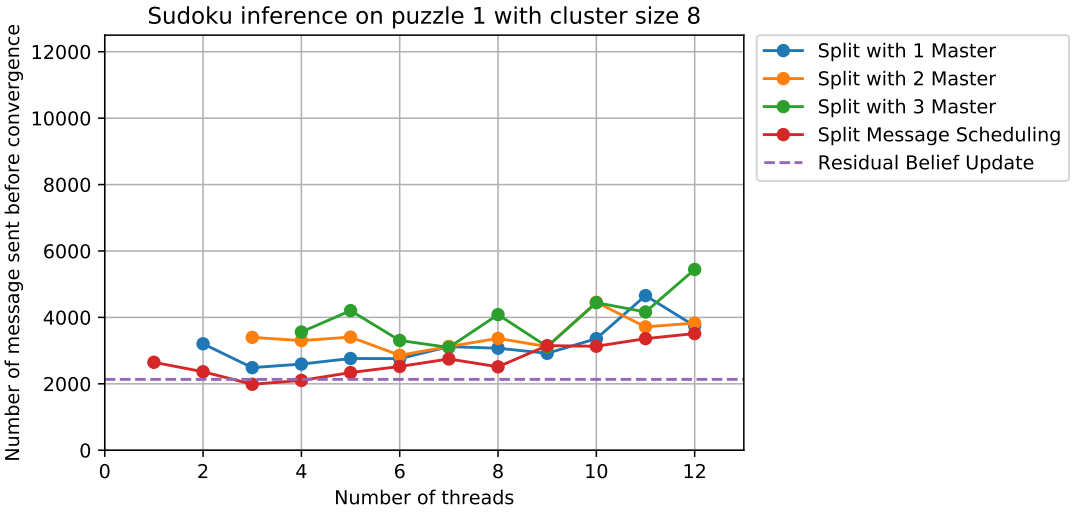


(b)

FIGURE 6.13: The speed-up relative to RBU and the number of messages sent before convergence for all the designed algorithms are shown for the sudoku solver using a cluster size of 8 RVs on puzzle 1.



(a)



(b)

FIGURE 6.14: The speed-up relative to RBU and the number of messages sent before convergence for the Split-MS variants are shown for the sudoku solver using a cluster size of 8 RVs on puzzle 1.

Overall, it is clear that the best algorithm to use is Split-MS. However, the larger the CG becomes, the more likely it is that Split-MS with multiple manager threads becomes useful. Nonetheless, the gain thereof is not enough to justify using the manager threads at this scale. If more threads were available, the manager threads might become useful.

One reason the speed-up of Split-MS reaches a plateau is the extra overhead Split-MS causes. This overhead is caused by having to store the old messages for absorption at a later stage. These old messages are locked as a single entity when a thread accesses it, causing more sequential code.

Another reason the speed-up of Split-MS (and all other algorithms) reaches a plateau is the shared message queue. While Split-MS with manager threads removes the problem of the shared message queue, it is still slower than Split-MS itself. This is because manager threads do not help with the work, reducing how parallelised the workload is.

As reported by [13], it would seem that parallel splash belief propagation (PSBP) does not plateau as early as our algorithms. It appears that no plateau was reached at 16 threads, which is the maximum number of threads that they tested. The maximum speed-up that the PSBP algorithm achieved was between 12 and 14 times for different tests. The reason PSBP can gain a higher speed-up is due to the higher amount of work done for every pop and push to the queue compared to our version of RBU. Remember that PSBP updates the nodes in a factor graph (FG), not the edges (thus, there is a queue instead of a message queue). Therefore, for every pop, there are multiple messages being sent. This reduces dependence on the sequential part of the process (the push to/ pop from the queue).

Another reason for the PSBP speed-up is the fact that PSBP uses belief residuals instead of message residuals (as explained in 1.1.5). It is significantly more expensive to calculate belief residuals compared to message residuals, especially in the context of cluster graphs. Due to the significant time spent calculating belief residuals, the time spent in the queue is less significant.

Also note that the version of RBU used in [13] queues the nodes instead of the messages. This allows the algorithm to pass all the messages connected to a node for a single pop from the queue. This allows for fewer accesses to the queue. Thus, less time is spent in this sequential part of the code compared to the time spent passing messages, leading to a greater speed-up.

With regards to the accuracy of the results, sudoku puzzles only have a single solution. Each of the algorithms solved the puzzles with the correct solution. Our best algorithms reached this single solution in fewer messages sent before convergence than the residual belief update algorithm. Finding the solution in fewer messages also hints that a more accurate solution can be found in cases where there is not a single solution, like in the satellite image denoising task. This suggests that there was no accuracy loss for any of the algorithms.

7 Conclusion and future work

7.1 Conclusions

In this thesis, we have designed four parallel, asynchronous message passing schedules (shown in Chapter 4) for use in the EMDW library (briefly discussed in Section 1.5). The purpose was to maximise the parallelisable code to gain the best speed-up over the benchmark. The benchmark was the residual belief update (RBU) algorithm, which was already available in the EMDW library.

The best of these algorithms was found to be split message scheduling (Split-MS). This technique splits the message passing (MP) into two parts (marginalising the source cluster to create the message; absorbing the message into the destination cluster). This allows for extra parallelisation, because threads only lock one cluster at a time, instead of two. This technique only pops a message from the message queue if the cluster it is connected to is not being written to by another thread. This eliminates the waiting time of threads due to memory clashes at clusters.

Even with the best algorithm (Split-MS), the speed-up reaches a plateau. This is due to the shared message queue and shared old messages, which require sequential access. This greatly reduces the parallelisability of the algorithms.

The best speed-ups measured (at plateau) during the tests were:

- Satellite image denoising: 5.3 times speed-up
- Sudoku solver with a cluster size of 7 RVs on puzzle 0: 3.3 times speed-up
- Sudoku solver with a cluster size of 8 RVs on puzzle 0: 2.7 times speed-up
- Sudoku solver with a cluster size of 7 RVs on puzzle 1: 4.0 times speed-up
- Sudoku solver with a cluster size of 8 RVs on puzzle 1: 2.0 times speed-up

The parallel splash belief propagation (PSBP) algorithm does not plateau so early. This is due to more messages being passed for every node popped from the queue (note that PSBP has a queue of nodes/factors instead of a message queue; it also uses factor graphs, instead of cluster graphs). This reduces the time spent popping from the queue, which reduces the sequential part of the process.

7.2 Future work

It is possible that Split-MS with one or more manager threads could outperform Split-MS without manager threads if given enough threads, because there are no plaeaus for these variants. Unfortunately, the computer we tested on only has 12 threads. Therefore, we suggest that some tests are run on a system with more threads available for parallelisation.

Split-MS can be sped up more if we sacrifice more memory. The current version of Split-MS has a single lock on the old messages as a whole. If we add a separate lock for every old message, extra parallelism is exposed.

Split-MS should be tested with a variant that has a different message queue for each thread. This way the worst bottleneck would be circumvented. This can theoretically

improve the speed-up of Split-MS.

Another possible solution to the bottleneck of Split-MS is to schedule only the clusters in the priority queue. The messages that need to be marginalised could then be added to a first-in-first-out (FIFO) queue. This will reduce the waiting time at the priority queue for the clusters and reduce the time to retrieve a message to marginalise.

To compare Split-MS and PSBP directly, PSBP should be adapted for use in the context of the EMDW library.

Distributed memory parallelisation should be explored. However, this solution has its own unique problems that will arise, as explained in Section 3.2.1.

A Denoised satellite images

This appendix shows the denoised satellite image results for each of the designed algorithms.

A.1 Noisy satellite image

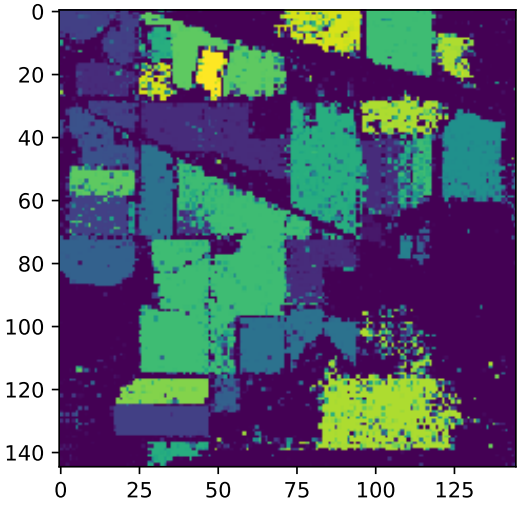


FIGURE A.1: Noisy satellite image used for denoising

A.2 Cleaned satellite image by SPush-MS

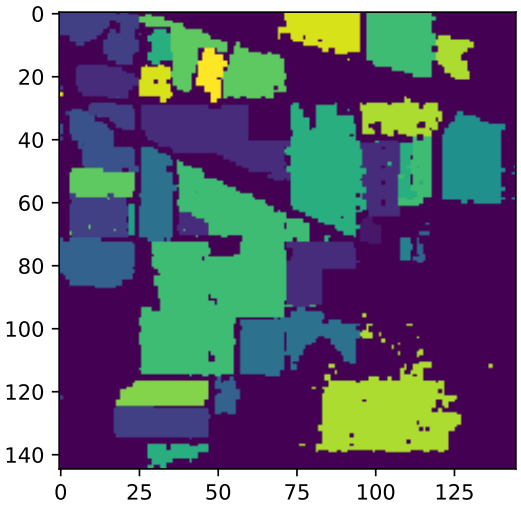


FIGURE A.2: Cleaned satellite image by SPush-MS.

A.3 Cleaned satellite image by MPush-MS

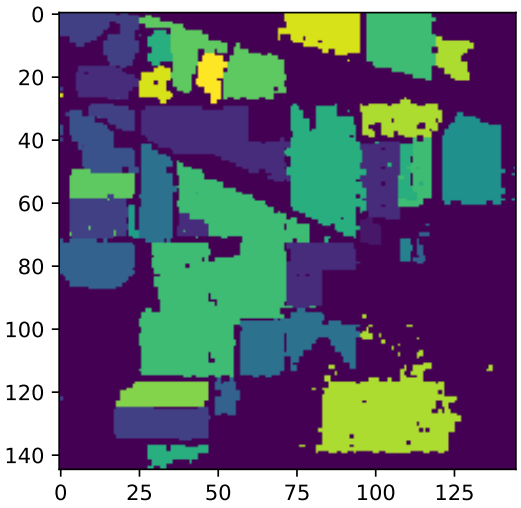


FIGURE A.3: Cleaned satellite image by MPush-MS.

A.4 Cleaned satellite image by Pull-MS

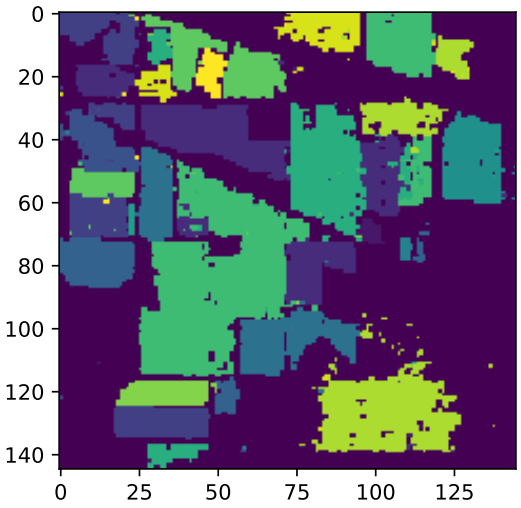


FIGURE A.4: Cleaned satellite image by Pull-MS.

A.5 Cleaned satellite image by Smart-MS

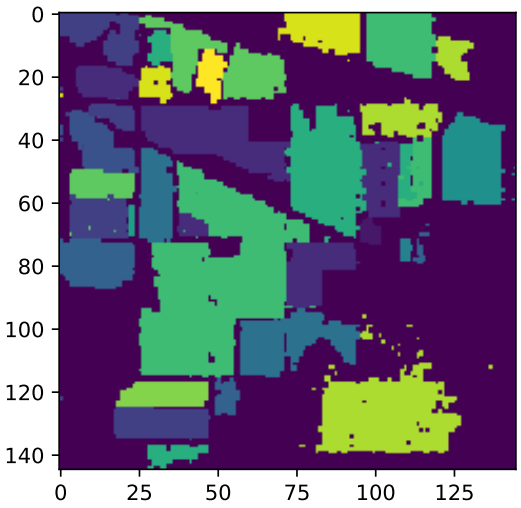


FIGURE A.5: Cleaned satellite image by Smart-MS.

A.6 Cleaned satellite image by Split-MS

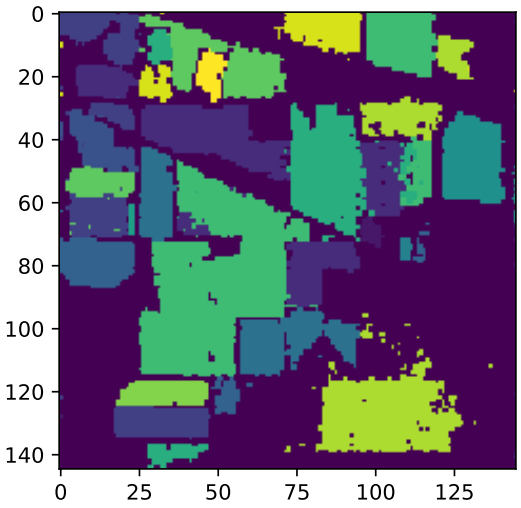


FIGURE A.6: Cleaned satellite image by Split-MS.

A.7 Cleaned satellite image by Split-MS with one manager thread

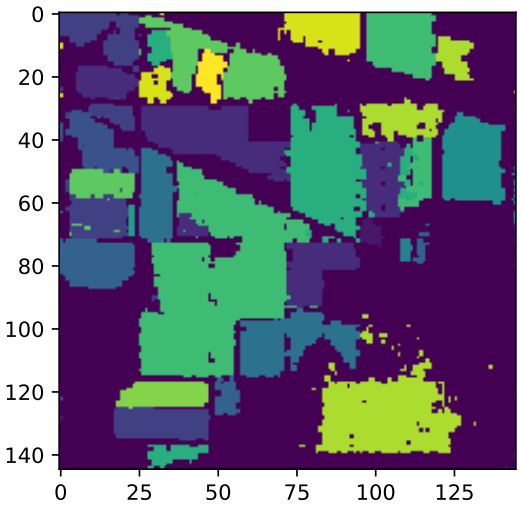


FIGURE A.7: Cleaned satellite image by Split-MS with one manager thread.

A.8 Cleaned satellite image by Split-MS with two manager threads

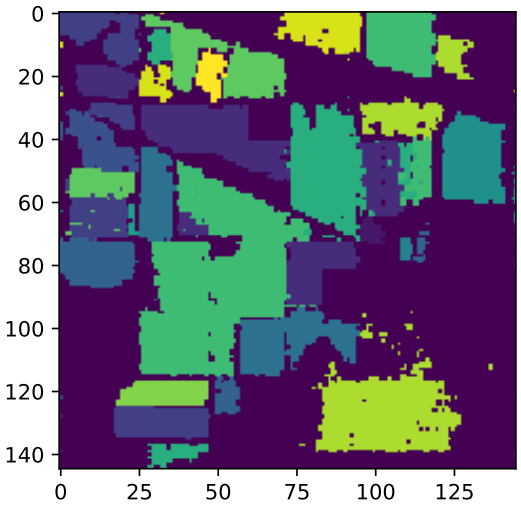


FIGURE A.8: Cleaned satellite image by Split-MS with two manager threads.

A.9 Cleaned satellite image by Split-MS with three manager threads

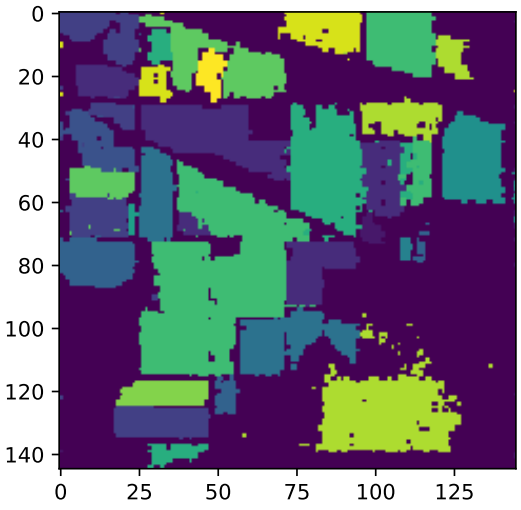


FIGURE A.9: Cleaned satellite image by Split-MS with three manager threads.

B CGs on which inference was tested

This appendix shows the graph structure of the CGs on which inference was done to test the algorithms that were designed for this work.

B.1 Sudoku CG with a cluster size of seven RVs

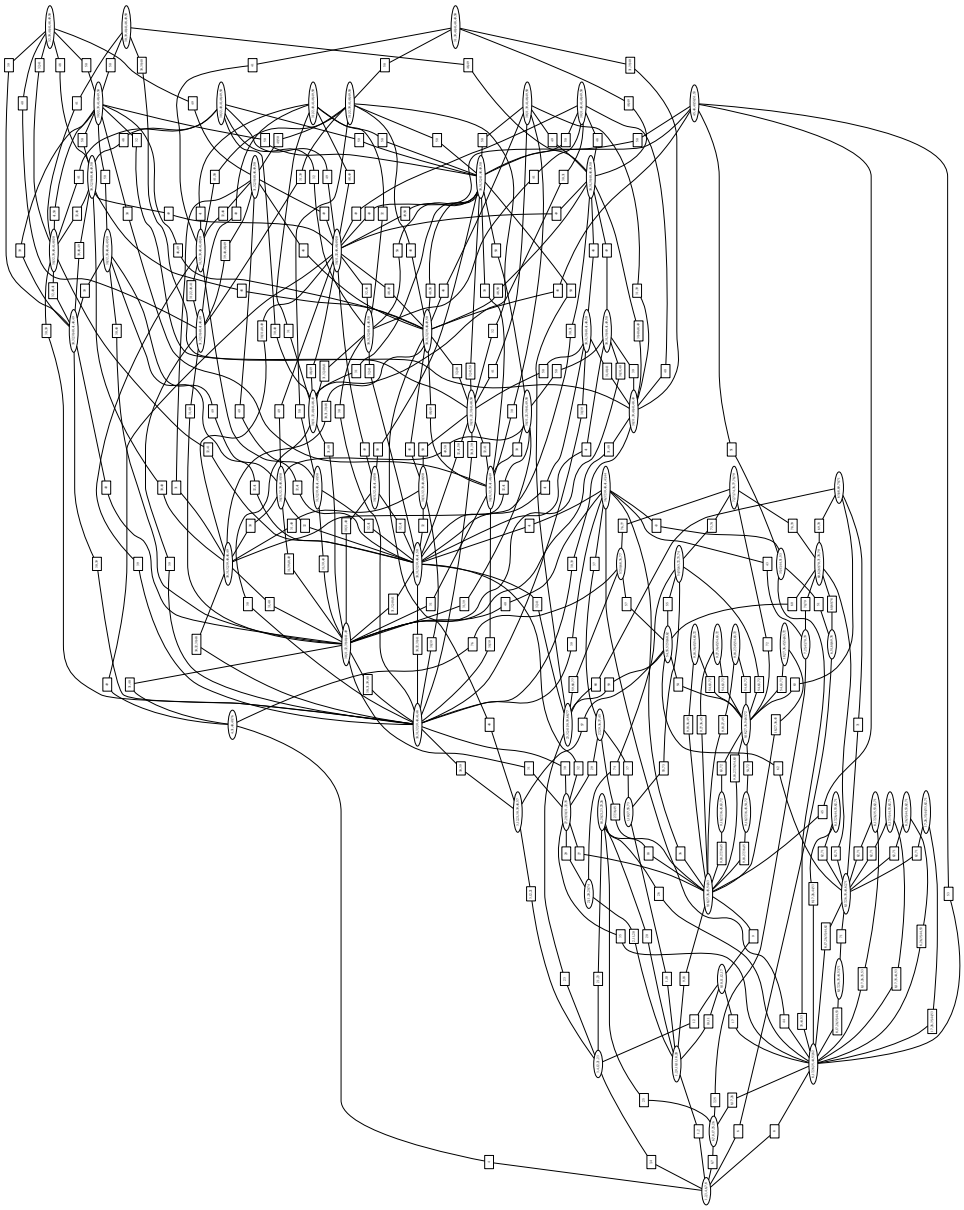


FIGURE B.1: Sudoku CG with a cluster size of seven RVs

B.2 Sudoku CG with a cluster size of eight RVs

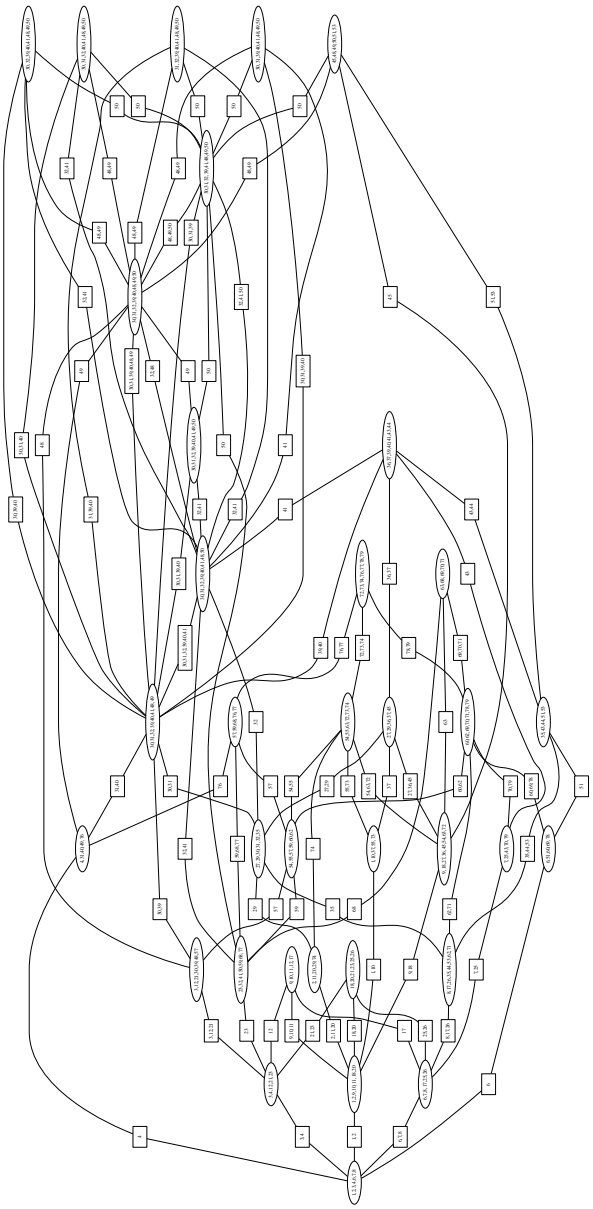


FIGURE B.2: Sudoku CG with a cluster size of eight RVs

Bibliography

[1] D. Koller and N. Friedman, *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009, ISBN: 978-0262013192.

[2] N. Friedman, *Inferring Cellular Networks Using Probabilistic Graphical Models*, 2004. DOI: 10.1126/science.1094068.

[3] D. J. Crandall, G. C. Fox, and J. D. Paden, "Layer-finding in radar echograms using probabilistic graphical models", in *Proceedings of the 21st International Conference on Pattern Recognition (ICPR2012)*, 2012, pp. 1530–1533.

[4] W. M. Dlamini, "A data mining approach to predictive vegetation mapping using probabilistic graphical models", *Ecological Informatics*, vol. 6, no. 2, pp. 111–124, 2011.

[5] B. Auslander, K. M. Gupta, and D. W. Aha, "Maritime threat detection using probabilistic graphical models", in *Twenty-Fifth International FLAIRS Conference*, 2012.

[6] D. Barber, *Bayesian reasoning and machine learning*. Cambridge University Press, 2012.

[7] P. Barry and P. Crowley, *Modern embedded computing: designing connected, pervasive, media-rich systems*. Elsevier, 2012.

[8] G. Elidan, I. McGraw, and D. Koller, "Residual belief propagation", in *Uncertainty in Artificial Intelligence (UAI)*, 2006, pp. 165–173.

[9] S Kullback and R. A. Leibler, "On Information and Sufficiency", *The Annals of Mathematical Statistics*, vol. 22, no. 1, pp. 79–86, 1951.

[10] P. P. Shenoy and G. Shafer, "Propagating Belief Functions with Local Computations", *IEEE Expert*, vol. 1, no. 3, pp. 43–52, 1986.

[11] S. L. Lauritzen and D. J. Spiegelhalter, "Local Computations with Probabilities on Graphical Structures and Their Application to Expert Systems", *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 50, no. 2, pp. 157–194, 1988.

[12] A.-M. Kermarrec, L. Massoulié, and A. J. Ganesh, "Probabilistic Reliable Dissemination in Large-Scale Systems", *IEEE Transactions on Parallel and Distributed systems*, vol. 14, no. 3, pp. 248–258, 2003.

[13] J. Gonzalez, Y. Low, and C. Guestrin, "Parallel Splash Belief Propagation", *Journal of Machine Learning Research*, vol. 1, pp. 1–48, 2009.

[14] T. Wittwer, *An Introduction to Parallel Programming*. 2006.

[15] P. S. Pacheco, *An Introduction to Parallel Programming*. Elserivier, 2011.

[16] R. H. Netzer and S. Ghosh, "Efficient race condition detection for shared-memory programs with post/wait synchronization", University of Wisconsin-Madison Department of Computer Sciences, Tech. Rep., 1992.

[17] N. M. Josuttis, *The C++ Standard Library Second Edition*. 2012.

[18] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 2010.

[19] D. R. Gene and M Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities", in *AFIPS*, ACM, 1967, pp. 483–485.

[20] M. D. Hill and M. R. Marty, "Amdahl's Law in the Multicore Era", *Computer*, vol. 41, no. 7, 2008.

[21] D. Koufaty and D. T. Marr, "Hyperthreading Technology in the Netburst Microarchitecture", *IEEE Micro*, vol. 23, no. 2, pp. 56–65, 2003.

[22] S. Saini, H. Jin, R. Hood, D. Barker, P. Mehrotra, and R. Biswas, "The Impact of Hyper-Threading on Processor Resource Utilization in Production Applications", in *2011 18th International Conference on High Performance Computing*, IEEE, 2011, pp. 1–10.

[23] T. Leng, R. Ali, J. Hsieh, V. Mashayekhi, and R. Rooholamini, "An Empirical Study of Hyper-Threading in High Performance Computing Clusters", *Linux HPC Revolution*, vol. 45, 2002.

[24] B. Chapman, G. Jost, and R. Van Der Pas, *Using OpenMP: portable shared memory parallel programming*. MIT press, 2008, vol. 10.

[25] G. S. Rodrigues, F. L. Kastensmidt, R. Reis, F. Rosa, and L. Ost, "Analyzing the impact of using pthreads versus OpenMP under fault injection in ARM Cortex-A9 dual-core", in *2016 16th European Conference on Radiation and Its Effects on Components and Systems (RADECS)*, 2016, pp. 1–6.